1981

Idiom matching: an optimization technique for an APL compiler

Feng Sheng Cheng
Iowa State University

CHENG, FENG SHENG

IDIOM MATCHING: AN OPTIMIZATION TECHNIQUE FOR AN APL COMPILER

Iowa State University PH.D. 1981

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Other _____

Idiom matching:
An optimization technique for an APL compiler

by

Feng Sheng Cheng

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1981

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I. INTRODUCTION

Previous Work

APL systems have been available for over ten years. Most implementations are generally referred to as "interpretive." In recent years, the use of APL has increased. This has led to the request for an APL compiler. But the execution of object code produced from a "straightforward" APL compiler is usually not faster than that of an interpretive system. This is because the object code produced from a straightforward compiler does not restrict the generality of the original source statement. For example, k+1 would still need run time checking to decide if k is a function or a variable. So the ideal compiler should embed various optimization techniques to produce simpler object code; and obviously this object code should preserve the original meaning of the original source program.

Delayed evaluation

Abrams (1) introduced the optimization technique of delayed evaluation for APL expressions. By delayed evaluation, we mean that the computation of intermediate results is deferred until the moment they are really needed. For example, to evaluate (A+B)[i], only A[i]+B[i] will be performed. Perlis (35) and Miller (30) proposed a "ladder machine" which will produce significant savings of temporary storage required to execute APL expressions. Their design is an extension of the work of Abrams.

Language restrictions

Compton (12) proposes a preprocessor which accepts APL-like statements and translates them into the corresponding PL/I statements. His design requires that APL users declare the shape and type of data at beginning of the program. Thus, not every APL program can be compiled. However, the execution time can be significantly reduced for those which are compilable. Jenkins (22) discusses the techniques for translating APL to ALGOL. His implementation indicates that the compiled code from a restricted APL program is significantly faster for scalar operations than an interpreted APL program.

The optimizing compiler presented in this thesis will not need language restrictions for APL. Roeder (40) has shown that a substantial amount of type determination can be performed at compile-time. Bauer and Saal (6) showed that 80 percent of the run time checking could be eliminated.

Compile-time optimization

Many optimizing techniques for compiling APL statements have been developed in recent years. The HP-3000 APL compiler (52) considers only one line at a time. It provides the concept of incremental compilation. Strawn (47) describes another important technique by which APL statements are partially parsed at compile-time and are completed at run time. He found that in a sample of programs only 2.4 percent of the identifiers and primitive functions were ambiguous. This suggests that his compile-time parsing technique is fairly practical.

Idiom matching

Idioms are programming language constructs used by programmers for the logical primitive operations for which no language primitives exist. In a functionally oriented language like APL idioms tend to be composite operations formed by composition of the primitive operations of the language.

Miller (30) has shown that the saving from optimizing idioms can be very impressive. Hoffmann (19) isolates a subclass of tree patterns and developed a matching algorithm for this class. Morris (32) implements a "phrase matcher" by converting from a regular tree grammar to a finite tree automaton.

A efficient solution to the idiom matching problem will directly provide the implementation with certain optimization techniques, e.g., elimination of redundant operators and constant propagation. This thesis will also show that the implementation of the algorithm for matching idioms can be generated systematically.

Tree automata

Tree automata provide the mechanism to implement the idiom matching function. A set of idioms can be preprocessed to form a finite tree automaton which will provide a matching algorithm with linear time complexity. Many researchers, such as Brainerd (9), Engelfriet (14) and Thatcher (50), have given definitions for tree automata. Here, we follow the notation of Thatcher in our discussion of tree regular expressions and tree automata. Brainerd also considers the generation of minimal tree automata. This thesis will develop a new algorithm to minimize the number of

states in a finite tree automaton. Our algorithm is an extension of the concept described by Hopcroft (20).

The Problem

The problem considered in this thesis is to design an idiom matching technique in a compiler for APL. Due to APL's richness of primitive functions, idioms tend to appear at the "expression level." APL users have to pay a very high price to execute their APL programs with operator-by-operator execution in conventional translators. However, if each idiom can be treated as a unit, this price can be significantly lowered.

Often APL expressions require large arrays for intermediate results in order to generate a final answer which is only a small array. By considering idioms as a unit for compilation, these intermediate results can often be avoided.

Thus, the purpose of idiom matching is to match idioms in source programs in order to generate very efficient target programs. The importance of idioms is that they are frequently used and often indicate considerable optimizations.

How idioms have been determined and how their corresponding code segments are generated are questions indepen-

dent of this work. This thesis will assume that some set of idioms has been collected. Perlis and Rugaber (36) have previously dealt with this question. In addition, Brown (10) has found 40 most frequently used idioms of height 2 in a sample of programs. This thesis investigates the possibility of matching APL expressions to a set of idioms. The problem of code segment generation is presently being considered by Omdahl (33).

Outline of Thesis

Chapter II begins with a discussion of the role that idiom matching techniques play in an optimizing compiler. It is shown that there are two problems in idiom matching: recognition and selection. The first problem, idiom recognition, can be solved by a tree automaton approach. The mechanism for constructing such a tree automaton from a regular tree expression for a set of idioms is then presented.

In Chapter III, an automata minimization algorithm is developed that obtains a time bound of $O(n^2 \cdot \log n)$ for any binary tree automaton.

Chapter IV deals with compile-time (as opposed to precompile-time) aspects of the problem. Emphases are placed on how idioms in an expression tree can be matched and what should be done if idioms are overlapped. It is shown that

the "best" non-overlapping idiom matches in the expression tree can be selected in O(n) time.

Finally, Chapter V summarizes these procedures and indicates some future extensions of the present work.

CHAPTER II. CONSTRUCTING AN IDIOM RECOGNIZER

Introduction

An overview of idiom matching is presented in this chapter. Then, tree automata are discussed with respect to solving idiom recognition problem. Emphasis is placed on how a tree automaton can be constructed from a regular tree expression.

We are dealing with compiler optimizations for very high level languages, such as APL. By very high level languages, we mean those languages that tend to involve less specification of algorithmic detail than the conventional high level languages, e.g., PASCAL and PL/I. The very high level languages specify "what" to do rather than "how" to do it. Thus, their compilers have a better chance to translate the source program to a more efficient target program.

Some work has been done in the area of compiler optimization for very high level languages, see Schwartz (43) and Roeder (40). Roeder studied the problem of type determination in APL.

Besides type determination, idiom matching appears to be an important technique for very high level language compilers even though it is usually not applicable to lower level languages. In a very high level language like APL, idioms are important because APL expressions usually incur a lot of computations. For example, one of the most frequently used branch composites in APL is

```
→ (v1 = v2) / v3 , v4 , v5.
```

This APL expression is similar to a PASCAL-like CASE statement that selects for execution a statement whose label is either v3, v4 or v5. An operator-by-operator execution would first perform two concatenation and one relational functions. Then, the compression function would generate another vector of which only the first element is needed by the branch. However, a more "intelligent" translator could produce the following equivalent PL/I statements:

```
IF v1(1) = v2(1) THEN GOTO v3;
IF v1(2) = v2(2) THEN GOTO v4;
IF v1(3) = v2(3) THEN GOTO v5;
```

In this example, many benefits can be obtained, such as saving object code, reducing temporary storage and decreasing execution time.

This thesis will assume that a set of idioms I1,..,Im has been collected and their corresponding code segments have been assigned. The first problem in idiom matching is to recognize all possible idioms in an expression tree E. It can easily be seen that some nodes in the same expression tree could be matched by more than one idiom. Thus, the second problem, idiom selection, is to select the "best" non-overlapping idioms in E. This will be discussed in detail in Chapter IV.

Definitions

The basic terminology of idiom matching in expression trees is given below:

Definition 1 (ALPHABET)

An alphabet SIGMA is a non-empty finite set of symbols.

Definition 2 (ARITY)

The arity of the node x in a tree, arity(x), is the number of descendants of x.

Definition 3 (RANKED ALPHABET)

An alphabet SIGMA is ranked if for each non-negative integer k a subset SIGMAk of SIGMA is specified, such that SIGMAk represents the elements of SIGMA with arity k; SIGMAk is non-empty for only a finite number of integers k.

Definition 4 (TREES)

The set of all trees over SIGMA, denoted by T-SIGMA, is the language over the alphabet SIGMA ∪ {(,)} defined recursively as follows,

1) If s belongs to SIGMA0, then s is in T-SIGMA.

2) If k ≥ 1, s in SIGMAk and t1,..,tk in T-SIGMA, then s(t1,..,tk) is in T-SIGMA.

Let V = {v0,..,vk} be a set of variable symbols, such that arity(vi) = 0 for each vi in V. The element v0 is usualy written as v. Let SIGMA' be defined as:

SIGMA'0 = SIGMA0 ∪ V and

SIGMA'k = SIGMAk for k > 0.

Thus, the set of all trees over SIGMA' is denoted by T-SIGMA'. The elements of SIGMA with zero arity are called constants.

Definition 5 (EXPRESSION TREE)

An expression tree E is any tree in T-SIGMA.

Definition 6 (IDIOM)

An idiom is any tree in T-SIGMA'.

Any idiom with variables v0,..,vk matches an expression tree E in T-SIGMA, with each vi matching some subtree in E.

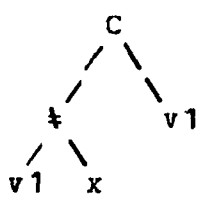
Repeated occurrences of any variable vi except v0 in the idiom must match identical subtrees in the expression tree. Thus, for each i > 0, vi is called a repeated variable. v0 (or v) is called an unrepeated variable. The match is defined, after Hoffmann (19), as

Definition 7 (MATCH)

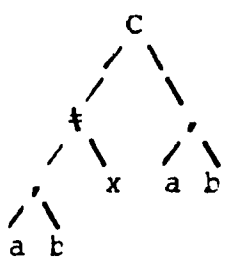
The idiom I matches the expression tree E at node p, if there are trees t1,..,tk in T-SIGMA such that substituting ti for each occurrence of vi in I, $1 \leq i \leq k$, and substituing certain trees in T-SIGMA for the occurrences of v0 in I, we obtain a tree I' equal to the subtree of E rooted at p.

Example 1

Idiom I



Expression E



For graphic reasons, the "C" represents /, the compression function. The APL idiom (v1⍀x)/v1 deletes all occurrences of the value associated with x in a vector v1. It matches the expression tree C(⍀(,(ab)x),(ab)) at the root C, with both occurrences of v1 matching the identical subtrees ,(ab).

The Idiom Recognition Problem

The first problem in idiom matching is to recognize all possible idioms in an expression tree. We now state this problem in a more formal fashion.

Idiom Recognition Problem:

> Given an expression tree E and a set of idioms $I=\{I1,\ldots,Ik\}$, locate in E all possible matches of Ii, $1 \leq i \leq k$.

A naive algorithm can solve the recognition problem with no repeated variables in $O(n \cdot m)$ time, where n is the size of expression tree and m is the sum of the idiom sizes. The concept behind this naive algorithm is the same as the naive string matching algorithm. A naive string algorithm searches each pattern at every possible start position in the expression string, abandoning the pattern whenever there is an unmatched character in that pattern. In the tree case, an expression tree that contains n nodes has $O(n)$ possible start positions that can be used to match every idiom. For each position, the match could be done by traversing each idiom. This traversal takes $O(m)$ steps in the worse case. Moreover, a recognition problem with repeated variables will take $O(n \cdot m) + O(n \cdot \log n)$ time. This can be seen from Figure 1. In Figure 1, the leftmost leaf a has been matched three

times. This number of matches is limited by the length of the path from root to that node, i.e., $O(\log_2 n)$ for a binary tree. Certainly, $O(n \cdot \log n)$ is the upper bound of the number of matches for repeated variables.

A modified algorithm with expected running time $O(n \cdot m)$ to solve the above problem can be constructed. The main concept behind this algorithm is to assign the same label to all roots of a common subtree in E, so that the labels can be used to decide whether their subtrees are identical. So the repeated variable problem can be solved at the roots of the subtrees that match repeated variables without traversing lower than those roots. This could be done by constructing a directed acyclic graph (DAG), which provides a good way of determining common subexpressions (3). Thus, the first step of this algorithm will construct a DAG from the expression tree with a label on each node. Then the second step needs only $O(n \cdot m)$ steps to find all possible matched idioms on the labeled expression tree.

The above algorithm basically solves the idiom matching problem for a finite set of idioms with an interpretive approach. On the other hand, the fundamental philosophy behind a fast matching algorithm is to preprocess the set of idioms. This is much the same idea as in the string pattern matching case. If the set of idioms is fixed and is to be

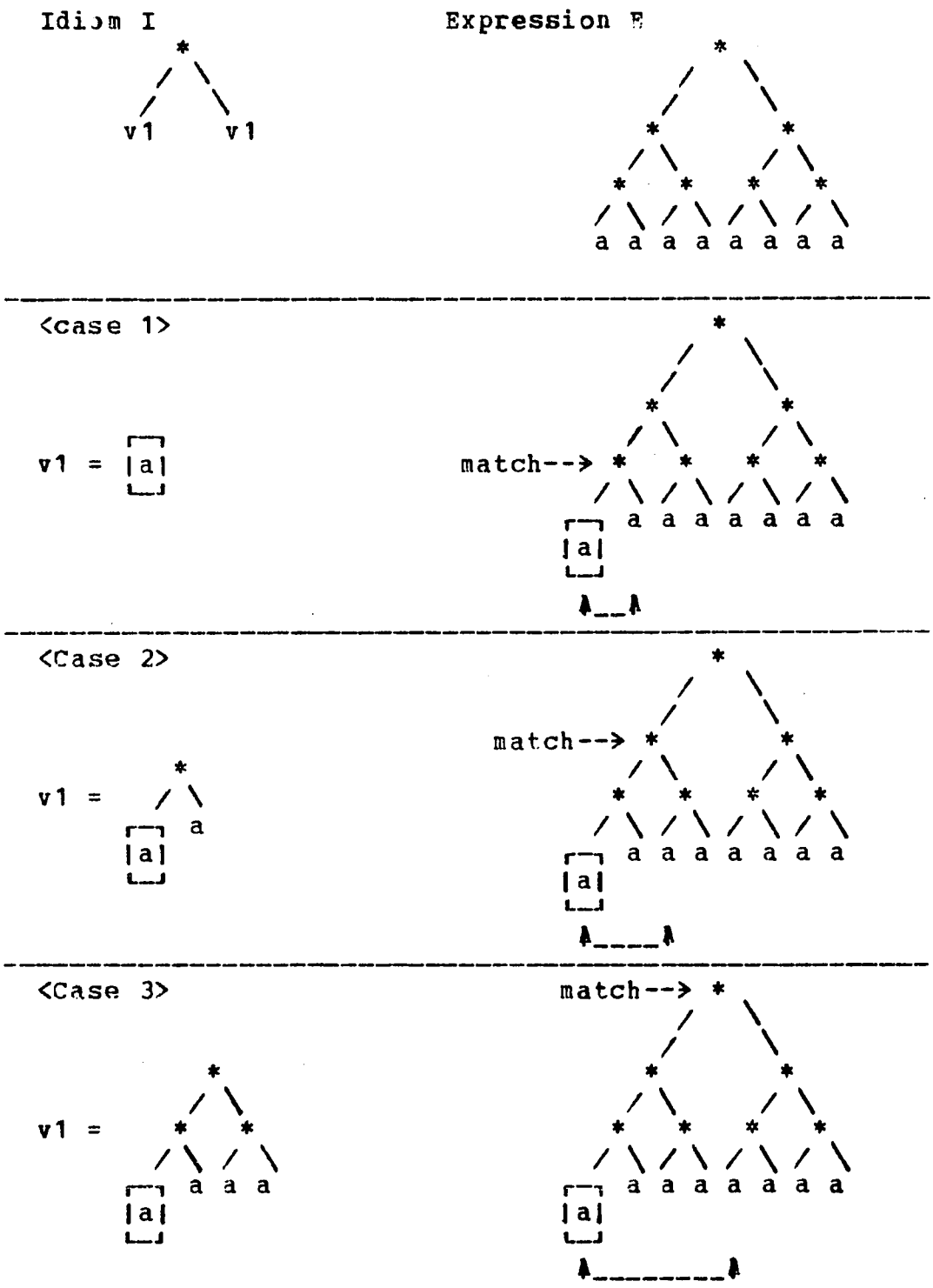


Figure 1. Three Matching Cases for the Leftmost Node a

matched against a number of expression trees, then it is advantageous to preprocess the idioms. Finite tree automata provide the mechanism for a linear idiom matching algorithm that makes exactly one state transition for each node in the expression tree. Thus, the time complexity of this idiom matching algorithm is O(n), not O(n•m), where n is the size of expression tree and m is the sum of the idiom sizes. Therefore, idiom preprocessing will be realized as the generation of a finite tree automaton.

In the following sections, tree automata are discussed with respect to solving the idiom recognition problem. A regular tree expression will be used to describe a given set of idioms. The number of idioms in this given set could be infinite. Thus, idiom preprocessing provides a more general solution to the idiom matching problem than the interpretive algorithm does. First, the definition of a regular tree expression will be introduced. The mechanism for constructing a tree automaton from a regular tree expression is then presented.

Regular Tree Expressions

The definition of the regular sets over a string alphabet is given in Kleene (24). Thatcher (50) considered the similar theory of regular sets of trees.

Definition 1 (OPERATIONS ON SETS OF TREES)

For $V \subseteq$ T-SIGMA, $W \subseteq$ T-SIGMA and a in SIGMA:

(1) the union of V and W is

$V \cup W$ = the union of the sets V and W

(2) the product of V and W at a is

$V \bullet_a W$ = the set of all trees obtained by replacing the frontier nodes labeled a in a tree from V with trees from W

(3) the closure of V at a is

$V^{*a} = \bigcup V^{ia}$ where $V^{0a} = \{a\}$ and $V^{(n+1)a} = V^{na} \cup (V \bullet_a V^{na})$.

Definition 2 (REGULAR SETS OF TREES)

The SIGMA-regular sets are the least class of subsets of T-SIGMA containing the singleton sets and closed under the operations $\cup$, $\bullet_a$ and $*_a$ for all a in SIGMA0. A set of trees is regular if it is SIGMA-regular for some SIGMA.

The regular expressions over T-SIGMA provide a syntactical mechanism for denoting regular sets of trees.

Definition 3 (REGULAR TREE EXPRESSIONS)

Regular expressions over T-SIGMA are defined recursively, as follows:

(1) For any element t in T-SIGMA, t is a regular expression denoting the tree regular set {t}.

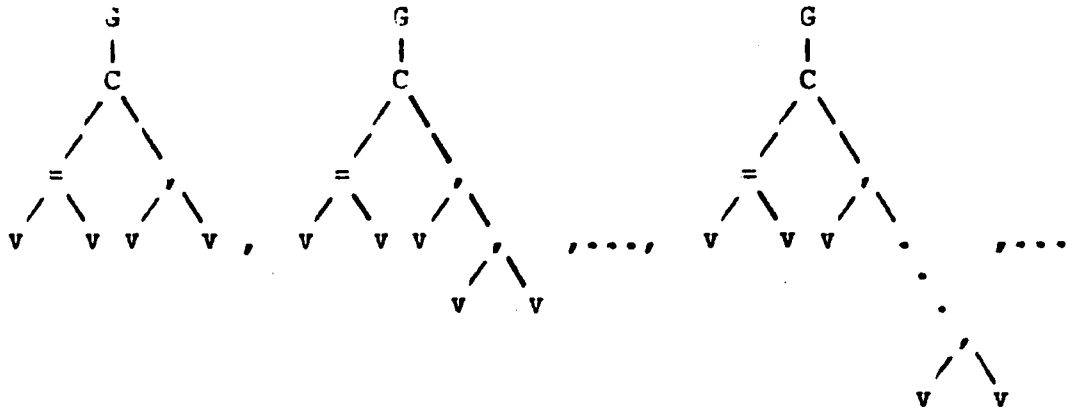
(2) If v and w are regular expressions denoting tree regular sets V and W, respectively, then:

  (a) (v U w) is a regular expression denoting V U W

  (b) (v •a w) is a regular expression denoting V •a W

  (c) (v*a) is a regular expression denoting V*a.

Since this work deals with a given set of idioms, the regular expressions for idioms are defined over T-SIGMA'. Note that the set V is a group of variable symbols and arity(vi) = 0 for each vi in V. For example, let S be the following set of APL idioms over {G,C,=,,} U V, where G and C represent the branch and compression functions, respectively.



The set S can be represented by the regular expression

(1) •u ((2)*u •u (3)) where (1) = G(C(=(vv) u))

(2) = ,(v u)

(3) = ,(v v).

This was found to be the most frequently used idiom in a sample of APL programs (10). Note that we have introduced a "dummy symbol u" which is assumed to be an element in SIGMA0.

Construction of Non-deterministic Tree Automata

Knuth (25) defines a binary tree as a finite set of nodes which either is empty or consists of a root and two disjoint binary trees. He has shown that there is a one-to-one correspondence between forests of trees and binary trees. Let all symbols in the alphabet be in SIGMA2 and SIGMA0 = {lambda}, the empty tree. If SIGMA consists of SIGMA2 and {lambda}, T-SIGMA represents the set of all binary tree over SIGMA U {lambda}. Therefore, the following

discussion can be simplified so that SIGMA consists of SIGMA2 only. From now on, T-SIGMA represents the set of all binary trees.

Definition 1 (TREE AUTOMATA)

A (deterministic) bottom-up finite tree automaton M over SIGMA is a system (N,T,n0,F) where

(1) N is a finite set of states;

(2) T is a transition function of the form:

$$T: SIGMA \times N \times N \longrightarrow N$$

(3) n0 in N is the initial state;

(4) $F \subseteq N$ is the set of final states.

A non-deterministic, bottom-up finite tree automaton is defined by allowing the range of the transition function to be the power set (set of subsets) of N. Now, consider the relation between regular expressions over T-SIGMA' and finite tree automata over SIGMA. In particular, every regular set of trees can be denoted by a regular expression and recognized by a non-deterministic, bottom-up tree automaton. Moreover, Thatcher (50) proves that every set recognized by a non-deterministic bottom-up tree automaton is also recognized by a deterministic bottom-up tree automaton. The following construction is a generalization of the construction of a non-deterministic automaton from a regular expression (3).

Theorem 1

For any regular set of trees in T-SIGMA' which is represented by a regular expression R, one can effectively construct an equivalent non-deterministic bottom-up tree automaton M over SIGMA.

Proof: The proof can be done by construction. A non-deterministic tree automaton M = (N,T,n0,F) can be constructed from R as follows:

(1) If R = t in T-SIGMA', then

(a) For each frontier node a in R, associate a unique state sa, and add the transition function value

T(a,n0,n0) = sa.

(b) For each variable "v" of V in R, associate the "don't care" state *.

(c) For each interior node b in R, associate a unique state sb and give a new transition function value

T(b,si,sj) = sb

where si is the state associated with the left son of b and sj is the state associated with the right son of b.

(d) Let F be the state associated with the root node in R.

(2) By the definition of regular expression, if R is not some t in T-SIGMA', then R is ri U rj or ri •a rj or ri*a for some choice of ri, rj and a. Assuming that there exist M' and M" such that M' = (N',T',n0,F') over SIGMA recognizes ri and M" = (N",T",n0,F") over SIGMA recognizes rj, we can construct M to recognize R. Assume N' and N" are disjoint except for n0 and those states associated with the nodes of zero arity in ri and rj.

(a) Union operation, ri U rj:

Let M = (N'UN",T,n0,F'UF") over SIGMA where for f in SIGMA,

i) if both s1 and s2 are in N', then T'(f,s1,s2) in T(f,s1,s2).

ii) if both s1 and s2 are in N", then T"(f,s1,s2) in T(f,s1,s2).

(b) Product operation, ri •a rj:

Let M = (N'UN",T,n0,F') over SIGMA, where for f in SIGMA,

i) if both s1 and s2 are in N', then T'(f,s1,s2) in T(f,s1,s2).

ii) if both s1 and s2 are in N", then T"(f,s1,s2) in T(f,s1,s2).

iii) For each a on the frontier of ri, let p be a's father and sa be the state associated with a. If a is a left son of p:

if x in T'(p,sa,s2), then

x in T(p,sm,s2) for sm in F";

similarily, x in T'(p,s1,sa) and sm in F" imply

x in T(p,s1,sm).

(c) Closure operation, ri*a:

Let M = (N',T,n0,F) over SIGMA. For each f in SIGMA, if both s1 and s2 are in N', then

T'(f,s1,s2) in T(f,s1,s2).

For each a with zero arity in ri, let p be a's father and let sa and sp be the states associated with a and p, respectively. If a is a left son of p, then generate a new transition function value

sp in T(p,sr,s2)

for sr in F' and for each s2 such that

sp in T'(p,sa,s2);

similarily, if a is a right son of p, then add

sp in T(p,s1,sr)

for sr in F' and for each s1 such that

sp in T'(p,s1,sa).

If both sons of p are a's, then

sp in T(p,sr,sr) for each

sp in T'(p,sa,sa).

Finally, F = {sa} U F'.

The usual induction argument about this construction proves that every regular set is recognizable.

If the product operator or the closure operator is used in a regular expression, then each final state can have a set of idioms associated with it. For example,

a(x) •x (b U c)

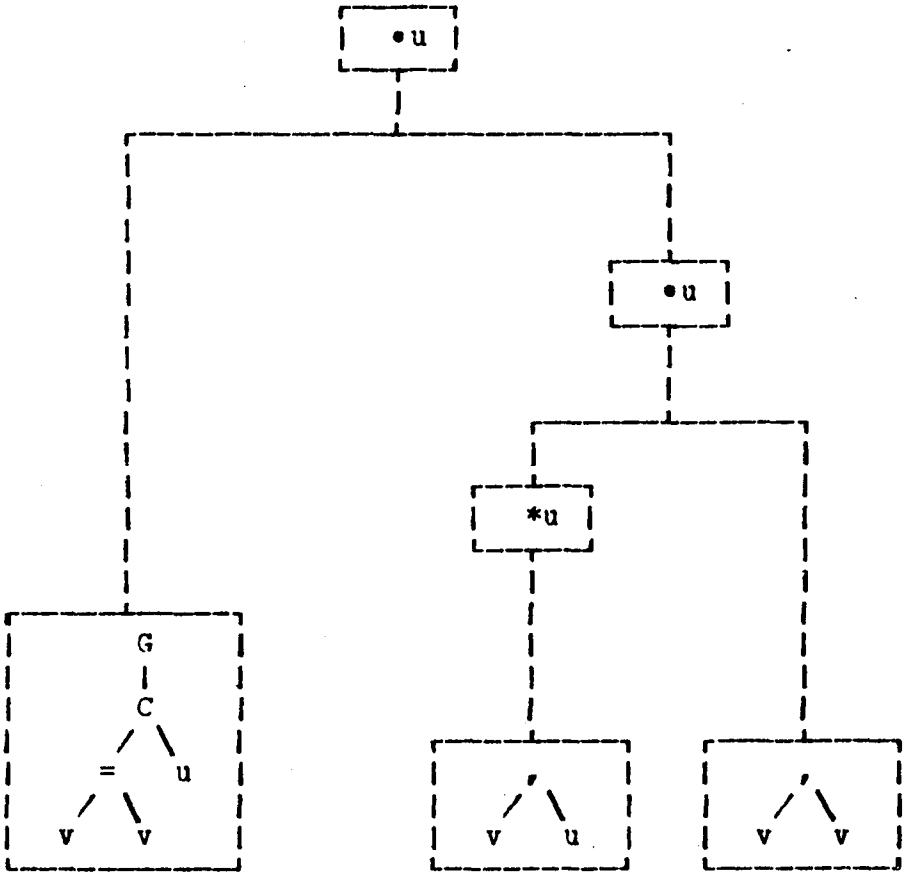
represents the set of idioms {a(b), a(c)} and yet there is only one final state in the automaton that recognizes this set. Thus, it is necessary that the semantic routine associated with a final state be able to distinguish between various members of the associated set of idioms.

Example 1

Consider the example in the last section of a regular expression

R = (1) •u ((2)*u •u (3))

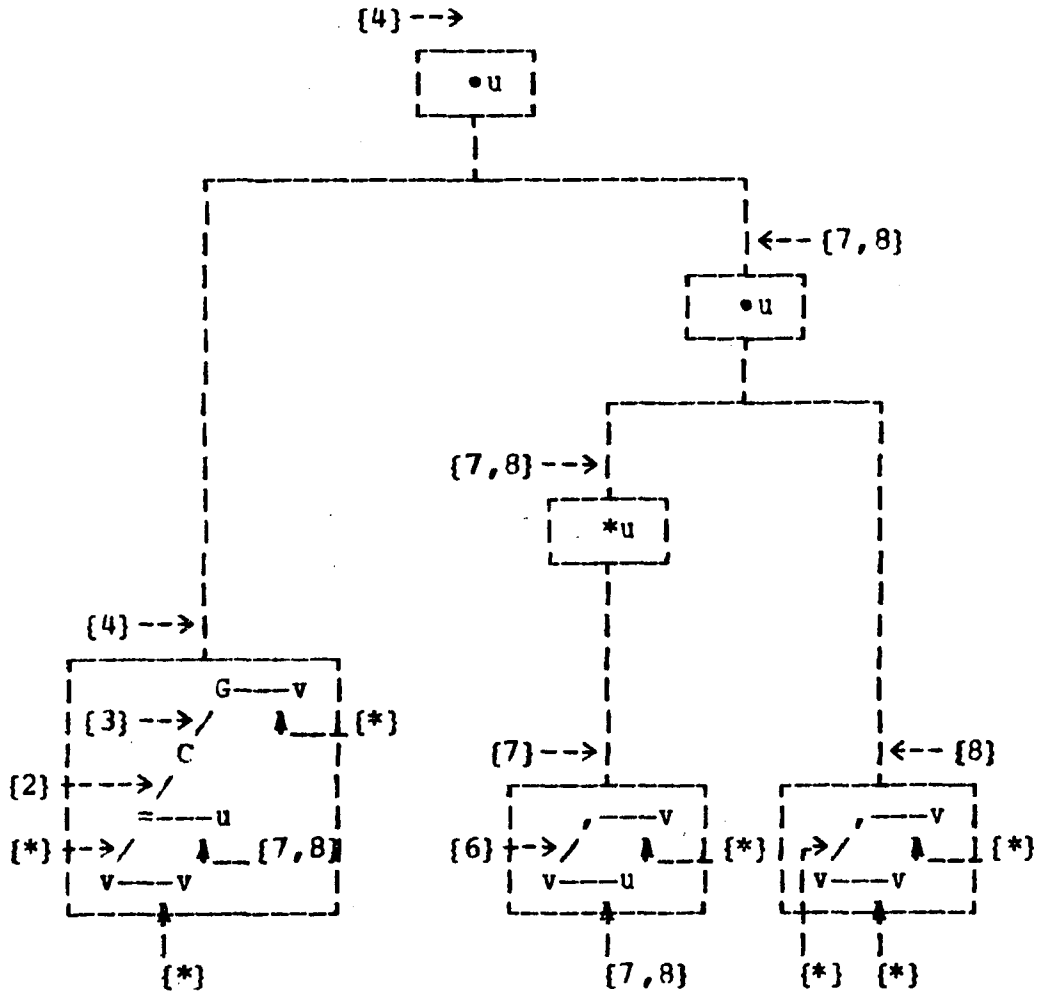
representing a set of idioms where (1) = G(C(=(vv) u)), (2) = ,(vu) and (3) = ,(vv). Figure 2 shows a graphic representation of R. The proof of Theorem 1 tells us how to construct from R an equivalent tree automaton M. Each node in R with its associated states is given in Figure 3 and M is the non-deterministic machine shown in Figure 4. Note we have introduced a "dummy variable v" in Figure 4 to indicate the right end of a regular subexpression. The root of the binary representation of an idiom with a dummy variable as its right son indicates this idiom can be matched throughout the expression tree. The execution of M is traced for the expression tree G(C(=(SS),(S,(SS)))) in Figure 5. In practice, all legal symbols should be considered when the corresponding automaton is being constructed. The S in Figure 5 denotes any such a symbol in input expression tree. In this particular example, we simply treat it as the symbol C and use the transition matrix of C to get the next state for S.



R = (1) •u ((2)*u •u (3))

where (1) = G(C(=(vv) u))
(2) = ,(v u)
(3) = ,(v v)

Figure 2. Graphic Representation of R



R = (1) •u ((2) *u •u (3))

where (1) = G(C(=(vv) u))
(2) = ,(v u)
(3) = ,(v v)

Figure 3. Graphic Representation of R with States

N={0,1,2,3,4,5,6,7,8}, n0={0},

F={4} and T=

|G|

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	6	6
1	0	0	0	0	0	0	0	6	6
2	0	0	0	0	0	0	0	6	6
3	4	4	4	4	4	4	4	4,6	4,6
4	0	0	0	0	0	0	0	6	6
5	0	0	0	0	0	0	0	6	6
6	0	0	0	0	0	0	0	6	6
7	0	0	0	0	0	0	0	6	6
8	0	0	0	0	0	0	0	6	6

|=|

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	2,6	2,6
1	0	0	0	0	0	0	0	2,6	2,6
2	0	0	0	0	0	0	0	2,6	2,6
3	0	0	0	0	0	0	0	2,6	2,6
4	0	0	0	0	0	0	0	2,6	2,6
5	0	0	0	0	0	0	0	2,6	2,6
6	0	0	0	0	0	0	0	2,6	2,6
7	0	0	0	0	0	0	0	2,6	2,6
8	0	0	0	0	0	0	0	2,6	2,6

|C|

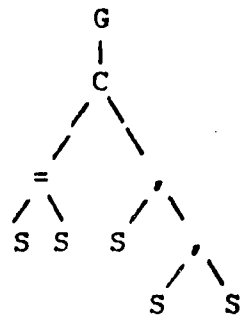
	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	6	6
1	0	0	0	0	0	0	0	6	6
2	3	0	0	0	0	0	0	6	6
3	0	0	0	0	0	0	0	6	6
4	0	0	0	0	0	0	0	6	6
5	0	0	0	0	0	0	0	6	6
6	0	0	0	0	0	0	0	6	6
7	0	0	0	0	0	0	0	6	6
8	0	0	0	0	0	0	0	6	6

|,|

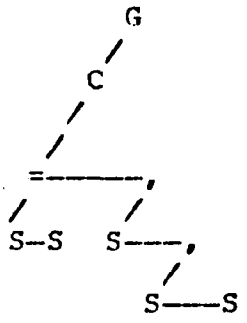
	0	1	2	3	4	5	6	7	8
0	8	8	8	8	8	8	8	6,8	6,8
1	8	8	8	8	8	8	8	6,8	6,8
2	8	8	8	8	8	8	8	6,8	6,8
3	8	8	8	8	8	8	8	6,8	6,8
4	8	8	8	8	8	8	8	6,8	6,8
5	8	8	8	8	8	8	8	6,8	6,8
6	7,8	7,8	7,8	7,8	7,8	7,8	7,8	6,7,8	6,7,8
7	8	8	8	8	8	8	8	6,8	6,8
8	8	8	8	8	8	8	8	6,8	6,8

Figure 4. A Non-deterministic Tree Automaton M

Input Expression Tree



Binary Representation



Step	Transition (Bottom-up)	Action
1	T(S,0,0) = {0}	
2	T(S,0,0) = {0}	
3	T(S,0,0) = {0}	
4	T(S,0,0) = {0}	
5	T(,,0,0) = {8}	
6	T(S,0,8) = {6}	
7	T(,,6,0) = {7,8}	
8	T(=,0,7) = {2,6}	
	T(=,0,8) = {2,6}	
9	T(C,2,0) = {3}	
	T(C,6,0) = {0}	
10	T(G,3,0) = {4}	Accepted
	T(G,0,0) = {0}	

Figure 5. M Accepting G(C(=(SS),(S,(SS))))

The tree automaton M is non-deterministic whenever there exist common subtrees in R. This is because we assign different next states to the same nodes in the common subtrees when M is being constructed from R. Note that any variable v in R could match a subtree rooted by any sym-

bol in the alphabet SIGMA, so that both ,(v,) and ,(vv) could match ,(G,) while the roots of ,(v,) and ,(vv) have been assigned two different states. Thus, we consider ,(v,) and ,(vv) in Figure 2 as common subtrees. If there are no common subtrees in R, then the tree automaton M constructed from R is deterministic.

Since a deterministic automaton is easier to simulate by a program than is a non-deterministic automaton, we would like to find a deterministic automaton accepting the same language for each non-deterministic automaton. Thatcher and Wright (51) have proved the equivalence of deterministic and non-deterministic automata by the subset construction. In next section, we will dicuss this transformation and its effectiveness.

Deterministic Tree Automata

The equivalence of non-deterministic and deterministic machines is well-known in conventional finite automata theory and the same equivalence exists in the theory of the tree automata. If a given set U in T-SIGMA is recognized by a non-deterministic finite tree automaton, then U is also recognizable by a deterministic finite tree automaton. This is done by a "subset construction" method. If a non-deterministic finite tree automaton (NFA) has n states, then the

number of states of the equivalent deterministic finite automaton (DFA) could in principle have 2^n states. In practice, only those subsets of the original states that are actually needed are generated. However, it is not necessary to construct a NFA separately as an input to the subset construction algorithm. Since the transitions of a binary tree automaton can be conviently represented by a "transition matrix," it is found that the transition matrix of a NFA is fully contained in the upper-left corner of the transition matrix of the equivalent DFA. Thus, each row or column of the upper-left corner of the DFA transition matrix represents a state in NFA. And this part of the matrix can be directly constructed from the regular expression. Each new state in the DFA is generated when a new set of states is found in the upper-left corner of the transition matrix. The entries of the transition matrix for these new states are the union of the entries of all corresponding states in the NFA (the upper-left corner). The process continues until no new state is generated.

The following algorithm constructs an eqivalent DFA from a given NFA.

procedure DFA;

Input: An NFA = (N,T,n0,F) over SIGMA where N is a set of states, T is the transition function, n0 is the initial state and F $\subseteq$ N is the set of final states

Output: A DFA = (N',T',n0,F') over SIGMA which accepts the same language

Comment: Let the number of states in N be j. And let N' be an extendable vector initialized such that: all states in N are stored in N'(0),...,N'(j-1); and each non-singleton subset of N that is a value of T is stored in N(j), N(j+1),...,N(k).

```
1.    function DFA_VALUE (X,Y: DFA states);
2.    begin
3.    for each symbol a in SIGMA do
4.        begin
5.        S := {s|T(a,ni,nj)=s for each ni in X, nj in Y};
6.        if S is not in N' then
7.             begin
8.             state_now := state_now + 1;
9.             N'(state_now) := S
10.            end
11.       T(a,X,Y) := S
```

```
12.          end
13.      end {of DFA_VALUE};
14.  begin {of DFA}
15.  j := the number of states in N;
16.  state_now := the number of states in N';
17.  while state_now > j do
18.      begin
19.      j := j + 1;
20.      for k = 0 to j-1 do
21.          call DFA_VALUE (N'(j),N'(k));
22.      for k = 0 to j do
23.          call DFA_VALUE (N'(k),N'(j))
24.      end
25.  F':={f'|f' in N' & f' contains any state that is in F}
26.  end {of DFA}
```

We illustrate this algorithm with the non-deterministic tree automaton in Figure 4. Beginning from line 15, j = 8 and state_now = 13. Enter the loop of lines 17-24. j = 9 in line 19. The function DFA_VALUE is called the first time with the parameters {6,8} and {0}. Line 11 is executed four times in this call, each for one input symbol. Four transitions are generated, T(G,9,0) = 0, T(C,9,0) = 0, T(=,9,0) = 0 and T(,,9,0) = 10. The loop of lines 20-21 calls the

function 9 times to fill the entries (9,0) (9,1) ... (9,8) of the transition matrix for each symbol. Similarily, the loop of lines 22-23 fill the entries (0,9) (1,9) ... (9,9) of the transition matrix for each symbol. Back to the loop of line 17, the condition "state_now > j" is still true. This loop is executed again to get the value for row 10 and column 10 of each transition matrix. If this process continues, we shall eventually get the result shown in Figure 6. Note there are four states, 1, 2, 5 and 7, which are unreachable from the initial state in Figure 6. In next section, a new minimization algorithm will detect any unreachable state easily.

In theory, a NFA with 9 states might generate a DFA with 512 states, one for each subset of the nine states. But, in the last example, we found that only 14 states (include 4 unreachable states) were needed. The "subset construction" which generates only actually needed states seems to be a very useful technique.

It is also noted that the DFA constructed might not be a "minimal" automaton. In next section, we will propose an algorithm to reduce the number of states for any DFA to a minimal number.

Rename: 9 = {6,8} 12 = {2,6}
10 = {7,8} 13 = {4,6}
11 = {6,7,8}

DFA M = (N, T, n0, F) where

N = {0,1,2,3,4,5,6,7,8,9,10,11,12,13},

n0 = {0}, F = {4,13} and T =

G

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	6	6	6	6	6	0	0
1	0	0	0	0	0	0	0	6	6	6	6	6	0	0
2	0	0	0	0	0	0	0	6	6	6	6	6	0	0
3	4	4	4	4	4	4	4	13	13	13	13	13	4	4
4	0	0	0	0	0	0	0	6	6	6	6	6	0	0
5	0	0	0	0	0	0	0	6	6	6	6	6	0	0
6	0	0	0	0	0	0	0	6	6	6	6	6	0	0
7	0	0	0	0	0	0	0	6	6	6	6	6	0	0
8	0	0	0	0	0	0	0	6	6	6	6	6	0	0
9	0	0	0	0	0	0	0	6	6	6	6	6	0	0
10	0	0	0	0	0	0	0	6	6	6	6	6	0	0
11	0	0	0	0	0	0	0	6	6	6	6	6	0	0
12	0	0	0	0	0	0	0	6	6	6	6	6	0	0
13	0	0	0	0	0	0	0	6	6	6	6	6	0	0

C

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	6	6	6	6	6	0	0
1	0	0	0	0	0	0	0	6	6	6	6	6	0	0
2	3	0	0	0	0	0	0	6	6	6	6	6	0	0
3	0	0	0	0	0	0	0	6	6	6	6	6	0	0
4	0	0	0	0	0	0	0	6	6	6	6	6	0	0
5	0	0	0	0	0	0	0	6	6	6	6	6	0	0
6	0	0	0	0	0	0	0	6	6	6	6	6	0	0
7	0	0	0	0	0	0	0	6	6	6	6	6	0	0
8	0	0	0	0	0	0	0	6	6	6	6	6	0	0
9	0	0	0	0	0	0	0	6	6	6	6	6	0	0
10	0	0	0	0	0	0	0	6	6	6	6	6	0	0
11	0	0	0	0	0	0	0	6	6	6	6	6	0	0
12	3	0	0	0	0	0	0	6	6	6	6	6	0	0
13	0	0	0	0	0	0	0	6	6	6	6	6	0	0

Figure 6. Constructing a DFA from an NFA

=

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	12	12	12	12	12	0	0
1	0	0	0	0	0	0	0	12	12	12	12	12	0	0
2	0	0	0	0	0	0	0	12	12	12	12	12	0	0
3	0	0	0	0	0	0	0	12	12	12	12	12	0	0
4	0	0	0	0	0	0	0	12	12	12	12	12	0	0
5	0	0	0	0	0	0	0	12	12	12	12	12	0	0
6	0	0	0	0	0	0	0	12	12	12	12	12	0	0
7	0	0	0	0	0	0	0	12	12	12	12	12	0	0
8	0	0	0	0	0	0	0	12	12	12	12	12	0	0
9	0	0	0	0	0	0	0	12	12	12	12	12	0	0
10	0	0	0	0	0	0	0	12	12	12	12	12	0	0
11	0	0	0	0	0	0	0	12	12	12	12	12	0	0
12	0	0	0	0	0	0	0	12	12	12	12	12	0	0
13	0	0	0	0	0	0	0	12	12	12	12	12	0	0

,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	8	8	8	8	8	8	8	9	9	9	9	9	8	8
1	8	8	8	8	8	8	8	9	9	9	9	9	8	8
2	8	8	8	8	8	8	8	9	9	9	9	9	8	8
3	8	8	8	8	8	8	8	9	9	9	9	9	8	8
4	8	8	8	8	8	8	8	9	9	9	9	9	8	8
5	8	8	8	8	8	8	8	9	9	9	9	9	8	8
6	10	10	10	10	10	10	10	11	11	11	11	11	10	10
7	8	8	8	8	8	8	8	9	9	9	9	9	8	8
8	8	8	8	8	8	8	8	9	9	9	9	9	8	8
9	10	10	10	10	10	10	10	11	11	11	11	11	10	10
10	8	8	8	8	8	8	8	9	9	9	9	9	8	8
11	10	10	10	10	10	10	10	11	11	11	11	11	10	10
12	10	10	10	10	10	10	10	11	11	11	11	11	10	10
13	10	10	10	10	10	10	10	11	11	11	11	11	10	10

Figure 6. (Continued)

CHAPTER III. MINIMIZING THE NUMBER OF STATES OF A TREE AUTOMATON

Introduction

A deterministic binary tree automaton has been constructed in Chapter II. In this chapter an automata minimization algorithm is developed that has a time bound of $O(n^2 \cdot \log n)$ for any binary tree automaton where n is the number of states. The conventional algorithm for minimizing the number of states in a finite string automaton needs $O(n^2)$ time. If we consider the minimization problem for tree automata with rank k symbols, as Brainerd (9) indicates, the execution time of such an algorithm is proportional to $O(n^{k+1})$. For finite automata with large numbers of states, this algorithm is inefficient. Hopcroft (20) proposes an $n \cdot \log n$ algorithm for minimizing states in a finite string automaton. In this thesis, we generalize his ideas to tree automata and show that the execution time of the minimization problem for any binary tree automaton is $O(n^2 \cdot \log n)$.

The present work is concerned with the minimization algorithm for finite tree automata and specially with the details of how such an algorithm could be used in practice. Without loss of generality, we already constructed a binary tree automaton over any ranked alphabet in Chapter II. Therefore, the input of the following minimization algorithm is assumed to be a binary tree automaton. However, it could be easily extended to n-ary tree automata.

Given a tree automaton M=(N,T,n0,F) over SIGMA, we want to find another tree automaton M' with the minimum number of states that is equivalent to M. As stated above, for any states x, y and input symbol a, T(a,x,y) denotes the next state of M. Figure 7 gives a simple example.

|a|

State x \ State y	1	2	3	4	5	6
1	1	1	1	1	1	2
2	1	1	1	1	1	3
3	1	1	1	1	1	4
4	1	1	1	1	1	5
5	1	1	1	1	1	6
6	1	1	1	1	1	6

|b|

State x \ State y	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	1	1	1	1	1

N={1,2,3,4,5,6}, n0={1}, F={6} and SIGMA={a,b}

Figure 7. Next State Tables T(a,x,y) and T(b,x,y)

A tree automaton can be converted into a minimal equivalent tree automaton by combining the equivalent states. State equivalence is defined formally by the following:

> States s and t are equivalent if and only if for each input string w, the following two conditions are fulfilled:
>
> (1) For any state y in N, T(w,s,y) is a final state if and only if T(w,t,y) is a final state.
>
> (2) For any state x in N, T(w,x,s) is a final state if and only if T(w,x,t) is a final state.

At the beginning, the set of states N could be partitioned into two blocks F and N - F. However, since the object of idiom matching is to perform idiom-directed translation in addition to simple recognition, it is necessary to place each final state in a separate block. Each final state indicates that one of a unique set of idioms has been found and a special routine for this set of idioms can be called to handle its semantic features.

The blocks of the initial partitions are then repeatedly split by examing the next states on a given input for all states in the block. States whose next states on a given input are in different blocks are not equivalent. When no

futher refinements are possible, states in the same block are equivalent and can be combined into one state.

Consider the example in Figure 7. The initial partition is (1,2,3,4,5)(6). On the first iteration we examine all next states of the states in the block (1,2,3,4,5). Since on input a, the next states of states 1, 2, 3 and 4 are all in the first block (1,2,3,4,5) and one of the next states of state 5, T(a,5,6), is in the second block (6), the first iteration refines the partition into the blocks (1,2,3,4), (5) and (6). On the second iteration, the block (1,2,3,4) is split into (1,2,3) and (4). It is seen that n iterations are needed before the final partition (1)(2)(3)(4)(5)(6) being reached. Because each iteration takes $O(n^2)$ steps, the total number of steps needed for this straightforward algorithm is $O(n^3)$.

To remove one state from a block, the above approach takes $O(n^2)$ steps. We now propose a new algorithm which also needs n iterations, but the worse case time bound is only $O(n^2 \cdot \log n)$. Before describing the new algorithm, we illustrate it by the example in Figure 7. First, the next state table T in Figure 7 is converted into a previous state table T^{-1} shown in Figure 8. T^{-1} is defined as $T^{-1}(s,a) = \{(x,y) \mid T(a,x,y)=s\}$ for a in SIGMA and s in N. The initial partition is still (1,2,3,4,5)(6). If we selected the block

(6) on input a, the previous state table T^{-1} tells us that state 5 is one of the previous states of the block (6) and those of other states 1, 2, 3 and 4 are not. Thus, the state 5 is different from the state 1, 2, 3 and 4 in block (1,2,3,4,5). The first iteration divides the block (1,2,3,4,5) into two subblocks (1,2,3,4) and (5). Note that the previous straightforward algorithm would have the same result after the first iteration being executed. But this new algorithm does not need $O(n^2)$ steps on the first iteration. The time needed to partition a block is proportional to the number of transitions into the block. The second iteration continues with the smaller subblock being selected. In this example, the subblock (5) is selected. Since the size of those selected subblocks are always less than half the size of the block which is being split, the total number of steps in the algorithm is bounded by $O(n^2 \cdot \log n)$.

One another important advantage of this new algorithm is that it is easy to detect any unreachable state from the initial state. The previous state table T^{-1} tells where each state comes from. For any non-initial state, if there is no previous state in T^{-1}, then it must be an unreachable state. And those non-initial states which have only unreachable states as previous states can't be reached from the initial state.

State #	Input symbol a	Input symbol b
1	(1,1) .. (1,5) (2,1) .. (2,5) (3,1) .. (3,5) (4,1) .. (4,5) (5,1) .. (5,5) (6,1) .. (6,5)	(1,1) .. (1,6) (2,1) .. (2,6) (3,1) .. (3,6) (4,1) .. (4,6) (5,1) .. (5,6) (6,1) .. (6,6)
2	(1,6)	-
3	(2,6)	-
4	(3,6)	-
5	(4,6)	-
6	(5,6) (6,6)	-

Figure 8. Previous State Table T^{-1}

Partition Algorithm

Let (N,T,n0,F) over SIGMA be a finite tree automaton where N is a finite set of states, T is a mapping from SIGMA X N X N into N, n0 is an initial state and F $\subseteq$ N is the set of final states. The algorithm for finding the equivalence classes of N is described below.

```
    procedure PARTITION1;
1.  begin
2.  Construct the previous state table T^-1;
3.  Delete any state which is unreachable from n0;
4.  Let initial partition be {B(1),..,B(p)};
5.  For each a in SIGMA, add all indices of the blocks in
       the initial partition to the vector L(a) except one
       block of the maximal size;
6.  while L(a) ≠ ∅ for any a in SIGMA do begin
7.     select any i from L(a) and delete it;
8.     for any B(j) such that the set of the previous
           states of B(i) includes any state in B(j) do
9.         begin
10.        Create a new block B(k);
11.        B(k) := {t|T(a,t,y) in B(i) or T(a,x,t) in B(i)
              with x, y in N, t in B(j)};
12.        B(j) := B(j) - B(k);  {partition B(j)}
13.        if j is in L(a) then
14.              add k to L(b), for each b in SIGMA
15.        else if |B(j)|<|B(k)| then
16.                add j to L(b), for each b in SIGMA
17.             else add k to L(b), for each b in SIGMA
18.        end  {of for loop}
19.    end  {of while loop}
20. end;  {of PARTITION1}
```

Example 2

Consider the tree automaton M in Figure 6. A previous state table T^{-1} can be constructed as in Figure 9. The notation (0-2,0-6) is a compact way of writing a number of state pairs and means (0,0) (1,0) (2,0) ... (2,6). Obviously, states 1, 2, 5 and 7 are unreachable states from the initial state 0. Since state 4 and state 13 indicate the same class of idioms, the initial partition is (0,3,6,8,9,10,11,12) (4,13). If block (4,13), which has fewer transitions into the block, is selected on input G, then the previous state pairs (3,4) and (3,13) in T^{-1} shows that state 3 is a previous state of the block (4,13). But $T^{-1}(\{4,3\},G)$ doesn't contain the pairs (x,4) or (x,13) where x is any state in the block (0,3,6,8,9,10,11,12) except 3. Thus, this block is divided into two subblocks (0,6,8,9,10,11,12) and (3). Next, if the block (3) and input C are selected, then (12,0) is the only pair that needs to be considered. This is because state 2 is an unreachable state so that (2,0) can be ignored. The pair (12,0) shows that the block (0,6,8,9,10,11,12) should be divided into (0) and (6,8,9,10,11,12). However, it also indicates the state 12 is different from any other state in (6,8,9,10,11,12). Thus, two new subblocks (12) and

(6,8,9,10,11) are generated. For the time being, the partition is (0) (3) (6,8,9,10,11) (12) (4,13). Again, if the block (12) is selected on input =, the pairs (0-13,7-11) show that state 6 is absent while states 8, 9, 10 and 11 are in. So, the block (6,8,9,10,11) is divided into two subblocks (6) and (8,9,10,11). Further iterations do not generate any new partition. So the final partition is (0) (3) (6) (8,9,10,11) (12) (4,13). Figure 10 shows the equivalent minimal machine M'. And Figure 11 shows the execution of this minimal machine for the same example presented in Figure 5.

The above algorithm omits some important implementation details. To keep $O(n^2 \cdot \log n)$ time, the algorithm needs certain data structures to reduce the computation. The following algorithm describes the details of the partition process so that the analysis of run time can be obtained.

Initially, for any input a the vector L(a) contains all indices of the blocks of the initial partition except one block of the maximal size. After a block B(i) is selected on input a, the index i is removed from L(a) and some block B(j) is split by the previous state pairs of B(i) and a. The algorithm terminates when all indices are removed from each vector L(a).

Input sym. / State #	G	C	=	,
0	(0-2,0-6) (4-13,0-6) (0-2,12-13) (4-13,12-13)	(0-1,0-6) (2,1-6) (12,1-6) (3-11,0-6) (13,0-6) (0-13,12-13)	(0-13,0-6) (0-13,12-13)	-
1	-	-	-	-
2	-	-	-	-
3	-	(2,0) (12,0)	-	-
4	(3,0-6) (3,12-13)	-	-	-
5	-	-	-	-
6	(0-2,7-11) (5-13,7-11)	(0-13,7-11)	-	-
7	-	-	-	-
8	-	-	-	(0-5,0-6) (0-5,12-13) (7-8,12-13) (10,12-13) (7-8,0-6) (10,0-6)
9	-	-	-	(0-5,7-11) (7-8,7-11) (10,7-11)
10	-	-	-	(6,0-6) (9,0-6) (6,12-13) (9,12-13) (11-13,0-6) (11-13,12-13)
11	-	-	-	(6,7-11) (9,7-11) (11-13,7-11)
12	-	-	(0-13,7-11)	-
13	(3,7-11)	-	-	-

Figure 9. The Previous State Table T^{-1} for M in Figure 6

R = (1) •u ((2)*u •u (3))

where (1) = G(C(=(vv) u))

(2) = ,(v u)

(3) = ,(v v)

Rename: 0 = {0} 3 = {8,9,10,11}

1 = {4,13} 4 = {12}

2 = {3} 5 = {6}

N={0,1,2,3,4,5}, n0={0}, F={1},

SIGMA={G,C,=,,} and T=

G	0	1	2	3	4	5
0	0	0	0	5	0	0
1	0	0	0	5	0	0
2	1	1	1	1	1	1
3	0	0	0	5	0	0
4	0	0	0	5	0	0
5	0	0	0	5	0	0

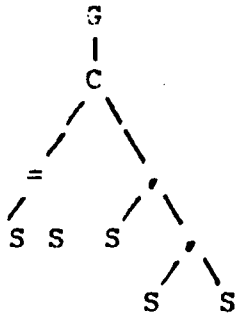
=	0	1	2	3	4	5
0	0	0	0	4	0	0
1	0	0	0	4	0	0
2	0	0	0	4	0	0
3	0	0	0	4	0	0
4	0	0	0	4	0	0
5	0	0	0	4	0	0

C	0	1	2	3	4	5
0	0	0	0	5	0	0
1	0	0	0	5	0	0
2	0	0	0	5	0	0
3	0	0	0	5	0	0
4	2	0	0	5	0	0
5	0	0	0	5	0	0

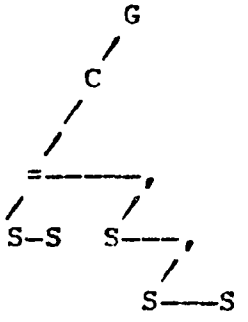
,	0	1	2	3	4	5
0	3	3	3	3	3	3
1	3	3	3	3	3	3
2	3	3	3	3	3	3
3	3	3	3	3	3	3
4	3	3	3	3	3	3
5	3	3	3	3	3	3

Figure 10. The Minimal Machine M'

Input Expression Tree



Binary Representation



Step	Transition (Bottom-up)	Action
1	T(S,0,0) = 0	
2	T(S,0,0) = 0	
3	T(S,0,0) = 0	
4	T(S,0,0) = 0	
5	T(,,0,0) = 3	
6	T(S,0,3) = 5	
7	T(,,5,0) = 3	
8	T(=,0,3) = 4	
9	T(C,4,0) = 2	
10	T(G,2,0) = 1	Accepted

Figure 11. M' Accepting G(C(=(SS),(S,(SS))))

procedure PARTITION2;

1. begin
2. For each a in SIGMA and each s in N construct
   $T^{-1}(s,a) := \{(x,y) \mid T(a,x,y) = s\}$;
3. Let initial partition be {B(1),..,B(h)}; k := h + 1;
   {each t in F could be assigned as a unique block}
4. if $T^{-1}(s,a) = \emptyset$ for each a in SIGMA and s $\neq$ n0 then
   s is a unreachable state;

```
5.   for each a in SIGMA do
6.      L(a) = {1,..,h} except i is the index of the largest
           subblock in the initial partition;
7.   while there exists an a in SIGMA such that L(a) ≠ ∅ do
8.      begin
9.      select a in SIGMA and i in L(a);
10.     L(a):=L(a)-{i};
11.     for each j < k such that |B(j)| > 1 and there exists
           a t in B(j) with T(a,t,y) in B(i) or T(a,x,t) in
           B(i), x,y in N, do
12.        begin
13.        B1(j):={t|T(a,t,y) in B(i) or T(a,x,t) in B(i)
              with x,y in N, t in B(j)};
14.        B2(j):=B(j)-B1(j);
15.        B1¹(j):=∅; B1²(j):=∅;
16.        for all T(a,x,y) in B(i) with x,y in B1(j) do
17.           add {x,y} to B1²(j);
18.        for any x,y in B1²(j) such that T(a,x,y) not in
              B(i) do
19.           begin
20.           B1²(j):=B1²(j)-{x,y};
21.           add {x,y} to B1¹(j)
22.           end;
23.        B1³(j):=B1(j)-B1¹(j)-B1²(j);
24.        r:=3; m:=3;
```

```
25.      for each y not in B(j) such that T(a,x,y) in B(i)
            with x in B1p(j), r≤p≤m do
26.         begin
27.         B:={t|T(a,t,y) in B(i) with t in B1p(j),
            r≤p≤m};
28.         m:=m+1; B1m(j):=B1p(j)-B; B1p(j):=B
29.         end;
30.      for each x not in B(j) such that T(a,x,y) in B(i)
            with y in B1p(j), r≤p≤m do
31.         begin
32.         B:={t|T(a,x,y) in B(i) with t in B1p(j),
            r≤p≤m};
33.         m:=m+1; B1m(j):=B1p(j)-B; B1p(j):=B
34.         end;
35.      if j is in L(a) then sw:=1 else sw:=0;
36.      max:=Max(|B2(j)|,|B1p(j)| 2≤p≤m)
37.      B(j):=B2(j);
38.      if |B(j)|≠max and sw=0 then
39.         for each b in SIGMA, add {j} to L(b)
40.      else sw:=1;
41.      for each t in B1¹(j) do
42.         begin
43.         B(k):={t};
44.         if |B(k)|=max and sw=0
45.            then sw:=1
```

```
46.          else for each b in SIGMA, add {k} to L(b);
47.          k:=k+1
48.          end;
49.       for p:=2 to m do
50.          begin
51.          B(k):=B1p(j);
52.          if |B(k)|=max and sw=0 then sw:=1
53.          else for each b in SIGMA, add {k} to L(b);
54.          k:=k+1
55.          end   {of for p loop}
56.       end   {of for j loop}
57.    end   {of while loop}
58. end.  {of PARTITION2}
```

Correctness of Algorithm

The first part of the work involved in showing the above algorithm correct is proving that the algorithm terminates. The algorithm must terminate since the only times that a block index is added into L(a) are at lines 6, 39, 46 and 53. If m is the size of the block being split, then there are at most m subblock indices being added to L(a) after the refinement of such a block. The number of refinements is at most n. Each time line 9 is executed, a block

index is removed from L(a). Thus, the algorithm must terminate.

The second part is to prove that two unequivalent states cannot be in the same block when the algorithm terminates. Assume two unequivalent states s and t are in block B(i) and one of the next states of s, T(a,s,w) or T(a,w,s) where w is any state in N, is in block B(j) and one of the next states of t, T(a,t,w) or T(a,w,t) where w is any state in N, is in block B(k) where $j \neq k$. Consider the point at which the block containing the next states of s and t is split in a manner that separates those next states. This is the first time that those next states of s and t are in separate subblocks. At this point, at least one of the two subblock indices is added to L(a). When this subblock index is selected from L(a), the block containing s and t is partitioned with s and t going into separate subblocks. Thus, s and t cannot both be in B(i).

Partition Algorithm Complexity

The straightforward algorithm for the partition problem requires time $O(n^3)$. With certain data structures, the above algorithm can be implemented in $O(n^2 \cdot \log n)$ time.

Let us consider in detail the implementation of the above algorithm. Lines 2-6 are executed only once. The

crucial part of the timing argument is to show that the loop of lines 7-57 can be executed in time proportional to $|T^{-1}(B(i),a)|$ which is the number of state transitions on input a terminating on states in B(i). Line 2 requires $O(n^2)$ steps. Lines 3-6 take only constant time. To find the appropriate j's at line 11, we need a vector BLOCKVEC such that BLOCKVEC(t) is the index of the block containing t. BLOCKVEC can be initialized in O(n) steps. Using BLOCKVEC, we can also construct a JLIST of all possible j's in line 11. By inspecting the inverse table $T^{-1}(B(i),a)$, we collect the t's such that T(a,t,y) in B(i) or T(a,x,t) in B(i). Each time a new t is found, the block containing t is located and is added to JLIST if it is not already in JLIST. Line 13 is executed at the same time JLIST is being constructed. t can be placed on a list of states to be split from the block j in JLIST. So, line 13 and line 14 require time proportional to the size of $B1(j)$.

The set $B1^2(j)$ can be constructed when B1(j) is being initalized. To execute lines 18-22 in constant time, we need a list NOMAT and a count field COUNT for each x in $B1^2(j)$. When adding {x,y} to $B1^2(j)$, we increase COUNT(x) by 1. If y is already in the list NOMAT(x), then y is deleted; otherwise, x is added to NOMAT(y). After $B1^2(j)$ is set up, $B1^1(j)$ can be constructed in constant time by check-

ing NOMAT for each x in $B1^2(j)$. So, lines 15-23 take $O(|B1(j)|)$ steps.

When line 11 is executed, a list YLIST containing every possible y in the state transition T(a,x,y) terminating on any state in B(i) is constructed. Then lines 24-29 can be done in $O(|B1^3(j)|)$ steps. If the same technique is applied to x in each T(a,x,y), then lines 30-34 can be done in $O(|B1^3(j)|)$ steps. Lines 35, 36, 38, 39 and 40 require only constant time. Line 37 needs $O(|B2(j)|)$ steps. Also, lines 41-48 and lines 49-55 require $O(|B1^1(j)|)$ and $O(|B1^2(j)| + |B1^3(j)|)$ steps, respectively.

Line 23 implies that B1(j) is the union of the subsets $B1^1(j)$, $B1^2(j)$ and $B1^3(j)$. We may conclude that the execution time of lines 12-55 is proportional to $O(|B1(j)|)$. But B1(j) is the intersection of B(j) and the set $\{t|T(a,t,y)$ in B(i) or $T(a,x,t)$ in B(i) with x,y in N$\}$. Since every partition of N contains only disjoint blocks, the aggregate time on all possible j's for the execution of lines 11-56 is $O(|T^{-1}(B(i),a)|)$. To complete the time complexity of this algorithm, the only thing remaining is the number of times the main loop is executed.

Let us consider the case of a state s, which is in a block not in L(a), but nevertheless has its block added to L(a). This can happen once at line 6. If this happens at

lines 39, 46 or 53, the block containing s is at most half the size of the block containing s prior to being split. We may conclude that the block containing s is not put into L(a) more than 1 + log n times. This means that s cannot be in the block i selected at line 9 more than 1 + log n times.

Suppose the execution time of the loop, lines 11-56, is proportional to $|T^{-1}(s,a)|$. The above discussion tells us that s cannot be in the selected B(i) more than O(log n) times. And we already knew the sum of $|T^{-1}(s,a)|$ for each s in N is n^2. So, the time complexity of this algorithm is seen to be $O(n^2 \cdot \log n)$.

In summary, lines 9-57 form the main loop which is executed at most O(log n) times for each symbol a in SIGMA. And we already knew that the execution time of lines 7-57 is $O(|T^{-1}(B(i),a)|)$, i.e., $O(n^2)$. Hence, the total time taken is $O(n^2 \cdot \log n)$.

CHAPTER IV. SELECTING AND MATCHING IDIOMS

Definition of Benefit Function

This chapter shows how idioms in an expression tree can be matched and what should be done if idioms are overlapped. Since idioms can be matched throughout the expression tree, some nodes may be in more than one match. For example, given the expression

G(C(=(xy) ,(,(ab) ,(cd))))

and three idioms

I1: ,(v ,(vv))

I2: ,(,(vv) ,(vv)) and

I3: G(C(=(vv) ,(v ,(vv))))

Figure 12 gives three possible matches. For graphic reasons, the branch fuction is typed as "G" and the compression function is typed as "C." I3 is one of the idioms defined in Figure 2. In Figure 12, there are two concatenation (,) functions that are matched by more than one idiom. If the subtree matched by I1 has been treated as a unit, then these two concatenation functions no longer exist, i.e., I2 and I3 can't match E, and vice versa.

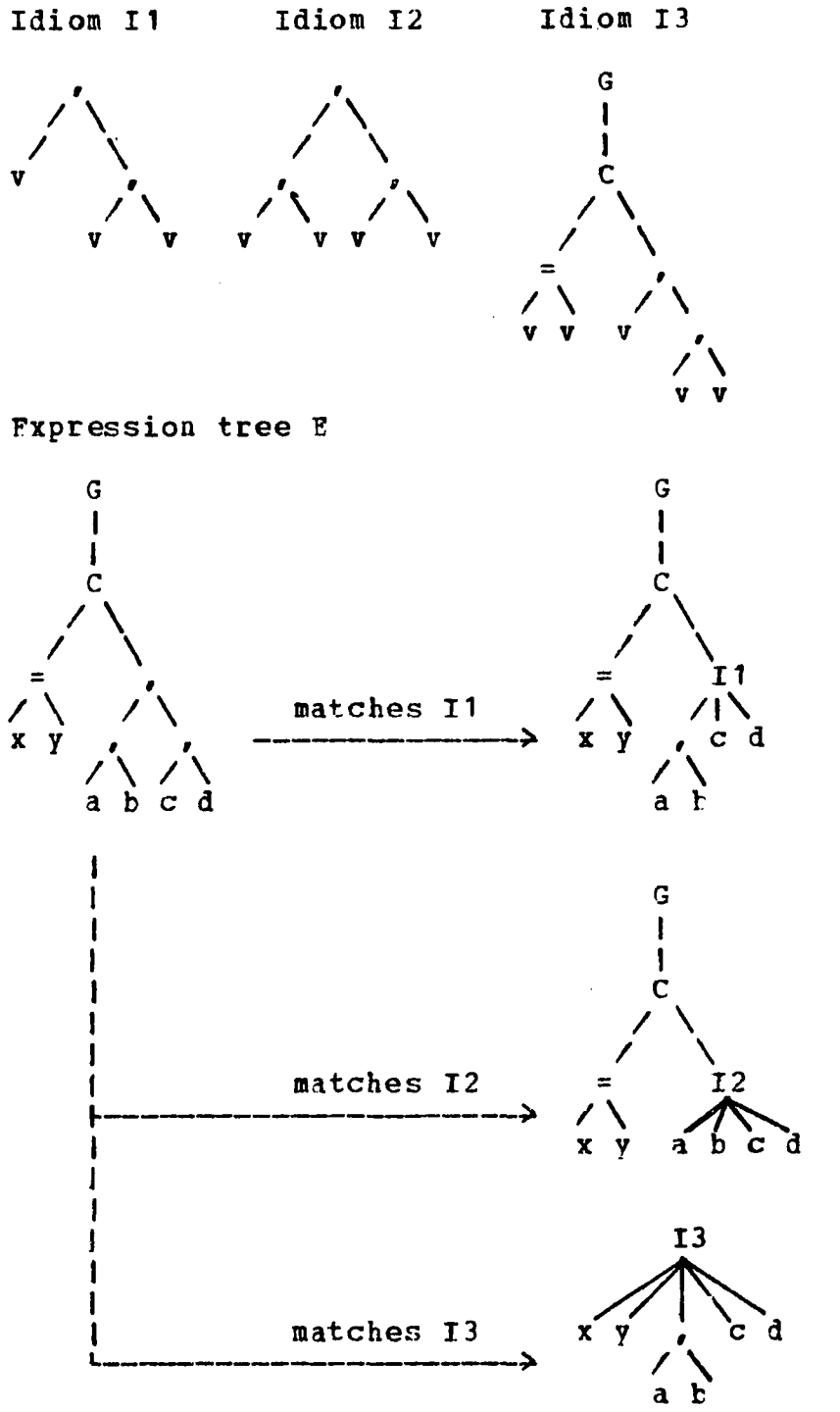


Figure 12. Three Overlapping Matches on an Expression Tree

Since we can't optimize all idioms at the same time, we should choose the "best" selection that will contain only non-overlapping matches on the expression tree. By the "best" one we mean the one which will gain the maximal benefit from optimization. This benefit could include saving of object code, decreasing execution time and/or reducing temporary storage. Then the benefit function of node x in E is defined as

$$\text{Benefit}(x) = \max_i \{ \text{BETA}(\text{IDi at } x) + \sum_{v \text{ in frontier of IDi}} \text{Benefit}(v) \}$$

where BETA(IDi at x) represents the benefit of IDi matched at node x and the v's are the roots of the subtrees in E that correspond to the variable leaves of IDi. If there is no match at x, then BETA(IDi at x)=0 and the v's are the immediate descendants of x. Note that the definition of BETA(IDi at x) allows different occurrences of the same idiom to have different benefits. For example, an idiom matched with array operands may have a larger benefit than the same idiom matched with scalar operands.

This simple definition also indicates the following important fact:

Benefit(x) $\geq$ sum of Benefit(immediate descendants of x).

This will guarantee that a bottom-up selection algorithm will always find the maximal benefit in a single traversal of the expression tree E. The number of the best matched idiom is saved at each node. Thus, the best non-overlapping matches in the subtree rooted by node x in E can be found by a top-down traversal of the subtree, ignoring matched idiom numbers in the bodies of the chosen idioms. So the final non-overlapping matches can be retrieved by a top-down traversal beginning from the root of the expression tree.

The Linear Idiom Selection Algorithm

Idiom Selection Problem:

> Given the set of all possible matches M in E, find the non-overlapping matches M', M' $\subseteq$ M, with the maximal benefit.

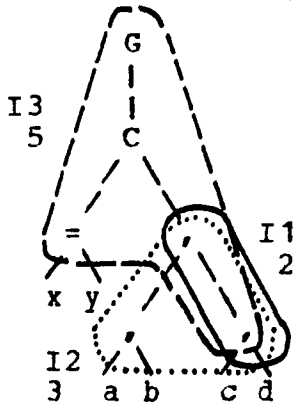
This problem can be solved by two simple linear algorithms. The first algorithm MARK is implemented by directly interpreting the definition of the benefit function. Although both top-down and bottom-up strategies can be used, we choose the latter for the reason that it can be combined into one pass with the recognition algorithm. The second, SELECTION, obtains the non-overlapping matches by a top-down traversal of the expression tree.

```
procedure MARK;

   Input:  An expression tree with the matched idiom
           numbers at each node
   Output: Maximum benefit and marked idiom at each node
   type    node = record
                  Op : operator or constant;
                  Match : list of matched idiom numbers;
                  Benefit, Mark  : integer
                  end;
   var     N : node;
1. begin
2. Traverse E in postorder, for each node N do
3.    begin
4.    N.Mark := 0;
5.    N.Benefit := sum of v.Benefit;  {v is a son of N}
6.    for each IDi in N.Match do {IDi that matches E at N}
7.       if N.Benefit < BETA(IDi,N) + sum of w.Benefit
            {each w is a root of a subtree in E that cor-
            responds to a variable leaf of IDi}
8.          then begin
9.               N.Benefit:=BETA(IDi,N)+sum of w.Benefit;
10.              N.Mark := IDi
11.              end;
12.   end  {of for N loop}
13. end;  {of MARK}
```

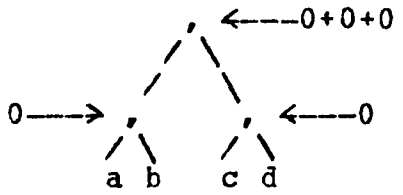
Example 2

Consider the expression tree E and the idioms I1, I2 and I3 in Figure 12. Using the simple benefit function BETA(IDi,N) = |IDi|, E will be matched as follows.



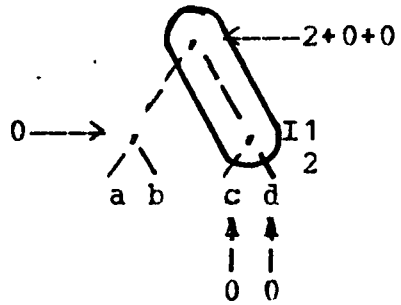
There are three cases at the node "," that is the right son of node C:

<Case 1>



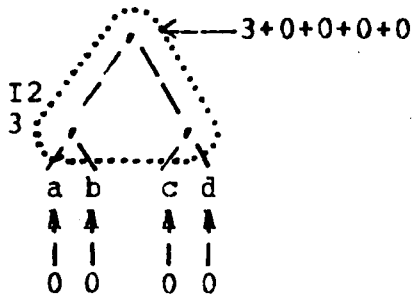
no mark

<Case 2>



mark I1

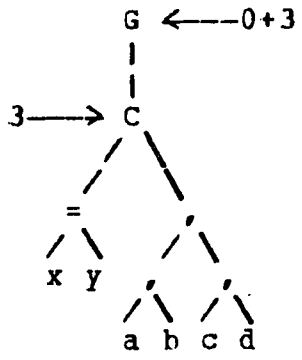
<Case 3>



mark I2

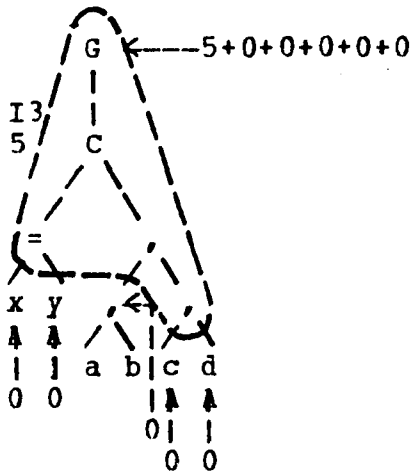
The case 3 with the largest benefit 3 would be saved at root , of the subtree of E. Then, root G of E is processed. There are two cases:

<Case 1>



no mark

<Case 2>

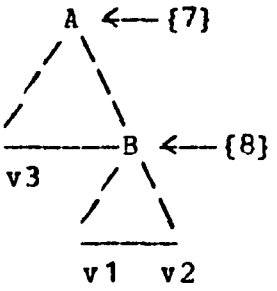


mark I3

Of course, the benefit 5 in Case 2 is saved at root .

Line 6 of the above algorithm assumes the actual shape of the matched idiom is known. In practice, the following implementation technique is needed to decide what this shape is. Each time a state that is associated with any operator (U, •a, *a) in the regular expression is reached, the set of

paths to all its frontiers nodes should be created. For example, states 4, 7 and 8 are such states in the graphic representation of Figure 3. If the following expression tree is being processed by the machine of Figure 3,



the current state {7} has three paths (A,...,v3), (A,...,v1) and (A,...,v2). The notation (x,...,y) means this path is a sequence of nodes from node x to node y. And the notation vi indicates the location of the variable v in tree; it doesn't imply a repeated variable. The last two paths, (A,...,v1) and (A,...,v2), are created by concatenating:

(A,...,B) || (B,...,v1) and

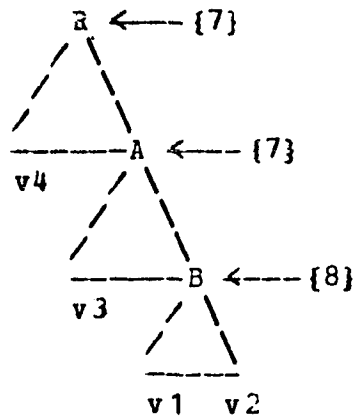
(A,...,B) || (B,...,v2).

Thus, each time a state associated with the closure operator *a is reached, the shape of the actual matched idiom is determined by the paths

(root,...,a) || all paths associated with the subtree that corresponds to a and

(root,...,b)

where b is any frontier node except a constant or "a". Let the above expression tree be one of the subtrees of the following expression tree,



The current state {7} has paths

(R,...,A) || { (A,...,v3),(a,...,v1),(A,...,v2) } and

(R,...,v4).

That is, (R,...,v4), (R,...,v3), (R,...,v2) and (R,...,v1). The same technique applies to the product operation, too.

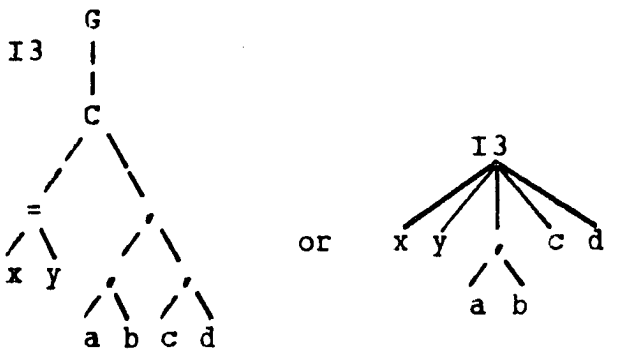
The bottom-up algorithm MARK marks each idiom that is the best match for the subtree rooted by x in E. Nevertheless, the marked idioms may not be included in the final solution to the idiom selection problem for E. The final solution is determined by the following algorithm. The algorithm starts at root of an expression tree. Then it traverses the expression tree top-down and obtains the selected idiom numbers of the best non-overlapping matches.

```
procedure SELECTION(N:item);

   Input:   An expression tree with the marked idiom
            numbers at each node
   Output:  Selected idiom numbers
   type   item = record
                 Op : operator or constant;
                 Benefit, Mark : integer
                 end;
   var    D : item;
1. begin
2. if N.Mark ≠0 then
3.    begin
4.    i := N.Mark;
5.    output {IDi matches at N};
6.    for each root D of the subtree that corresponds to a
           variable leaf of IDi do
           {left-to-right, preorder traversal}
7.         call SELECTION(D)
8.    end
9. else for each descendant D of N do
10.         begin {left-to-right, preorder traversal}
11.         output D;
12.         call SELECTION(D)
13.         end
14. end;  {of SELECTION}
```

Example 3

Consider the expression of Example 2. The final non-overlapping matches are shown below.



An Idiom Matching Machine

This section describes an idiom matching machine that locates the matches with the maximal benefit in an expression tree. The idiom matching machine here will take advantage of the preprocessed idiom set to achieve the fastest matching performance. The machine that uses a minimal tree automaton will make only one state transition for each node in an expression tree.

When a tree automaton is being constructed, all variables (include repeated variables) are associated with the "don't care" state. The transition function of an automaton can't tell the difference between the repeated variables and the unrepeated variables. The repeated variables are checked only after a final state has been reached.

Let all the roots of a common subtree in an input expression be assigned the same label. By checking the labels, the repeated variable problem can be solved at the roots of the subtrees that match repeated variable without traversing lower than those roots. Thus, a directed acyclic graph (DAG), which provides a good way of determining common subexpressions, could be constructed (in O(n) time) for each input expression tree. Aho gives such an algorithm in (3).

As shown above, each node associated with a final state contains path information that determines what the shape of matched idiom is. The path information associated with each final state can be used to locate every variable. Thus, the question of whether a repeated variable vi matches the same subtree that the other occurrences of variable vi match can be answered by checking the labels of the corresponding descendants. The following algorithm summarizes the behavior of an idiom matching machine.

procedure IDIOMACHINE;

Input: An expression tree E and the minimal automaton M which was defined in the last section

Output: The non-overlapping matches with the maximal benefit

```
1.  begin
2.  let initial state be n0;
        {the state of an empty binary tree is n0}
3.  Traverse E in postorder, for each node M in E do
4.     begin
5.     STATE = T(M,s1,s2) where s1 is the state of M's left
              son and s2 is the state of M's right son;
6.     if STATE is in F then
7.           begin
8.           for each idiom I of F do
9.               if the repeated variables of I are not
                    matched then
10.                 remove I from Match list;
11.          using the statements in the body of MARK algor-
                ithm, mark the node with the maximal benefit
                at M
12.          end
13.    end  {of for M loop}
14. call SELECTION (root of E);
        {the best non-overlapping matches can now be traced
        top-down from the root of E}
15. end.  {of IDIOMACHINE}
```

One question that might remain is what the variables of a closure operation mean. They could mean that every occurrence of the corresponding trees of these variables should have the same label (as repeated variables). On the other hand, they could simply be treated as unrepeated variables. Since each final state in an automaton either represents a single idiom or represents a class of idioms which have some common features, each final state knows all locations of the variable leaves of its corresponding idioms. For each variable vi, where i > 0, in a regular expression, the checking of repeated variable must be performed; for v0, path information is collected only for use in computing the benefit function.

CHAPTER V. CONCLUSIONS

High Level Optimization

This thesis investigated the design of an idiom matching technique in a compiler for APL. An idiom is essentially a term with variables that can be compiled as a unit to produce efficient programs. Idiom matching is the process by which a source program's parse tree can be scanned to locate occurrences of idioms, which can subsequently be used to produce optimized code. This optimization technique is particularly useful in APL. Two problems in idiom matching, recognition and selection, have been defined and solved.

The recognition problem deals with a technique to recognize occurrences of an idiom in an input tree. The recognition problem has been solved with a tree automaton approach. The set of idioms is preprocessed to get a fast matching algorithm that takes only one state transition for each node in an input expression tree. A practical automaton minimization algorithm is developed that obtains a time bound of $O(n^2 \cdot \log n)$ for any binary tree automaton with n states. For those idioms with repeated variables (i.e., a

variable that occurs more than once in the idiom), the tree automaton approach matches by recognizing a unique label associated with each identical subtree. Moreover, an algorithm is provided to construct the tree automaton systematically. The tree automaton approach appears to be a good choice for the recognition algorithm since the idiom set is fixed in advance and is matched repeatedly by a number of expression trees.

Once all occurrences of idioms in an input tree are recognized, it is necessary to be able to select those which are to be used for subsequent optimization. This step is needed, particularly when two or more idioms overlap over some portion of the input tree. The selection problem deals with determining how to select the most beneficial idioms for maximum cost advantage by any sort of benefit function. To this end, we define a general purpose "benefit function." The cost benefit of an occurrence of an idiom may be calculated in any way for an optimization required, provided it satisfies certain general purpose criteria we define. The benefit function is versatile enough to allow the same idiom to have different values when matched at different places. The complete solution of the idiom matching problem has therefore been solved by a one-pass bottom-up algorithm followed by a one-pass top-down algorithm and requires only $O(n)$ time.

Future Work

Considering the huge size of the transition function of a DFA for a large set of idioms, some compactification technique must be applied before this approach is actually used for idiom matching. Figure 10 shows that most entries of the transition matrices are either constant or have constant columns and/or rows. Several table compactification techniques have been developed (3, 49). However, work is needed to determine the best technique in this case.

Although much improvement is expected from code generated in conjuction with the optimization technique presented in this thesis, experimental studies should be performed to confirm it. However, this can not be done until the appropriate code segments are constructed.

The current design will obviously cause error handling to be more complicated. Even though we assume that only "production" programs would be dealt with in this optimizing compiler, the debugging feature should still be an integral part of any APL system.

What are the criteria to define the idiom selection benefit function? Should it depend on operand type, size of object code, user-define optimization level or time-space tradeoff?

Idioms suggest optimizations and language extensions. It is clear that with more idioms in use, more savings would occur. This suggests that idioms should be widely encouraged for implementation as well as for stylistic reasons.

This thesis has dealt with idioms that are regular tree expressions. Nevertheless, futher research is needed to discover if more general expression classes are of interest, which could still maintain efficient matching.

There are numerous unanswered questions that remain ahead.

BIBLIOGRAPHY

1. Abrams, P. S. "An APL Machine." Stanford Electronics Lab Technical Report 3, 1970.

2. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. The Design and Analysis of Computer Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Co., 1974.

3. Aho, A. V. and Ullman, J. D. Principles of Compiler Design. Reading, Massachusetts: Addison-Wesley Publishing Co., 1977.

4. Aho, A. V. and Ullman, J. D. The Theory of Parsing, Translation and Compiling, Vol. II. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.

5. Ashcroft, E. A. "Towards an APL Compiler." Proceeding of the Sixth International APL User's Conference, Anaheim, California, 1974.

6. Bauer, A. M. and Saal, H. J. "Does APL Really Need Run-time Checking?" Software--Practice and Experience 4 (1974): 129-138.

7. Bingham, H. W. "Content Analysis of APL Defined Functions." APL 75, Pisa, Italy, 1975.

8. Bingham, H. W. "Dynamic Usage of APL Primitive Functions." APL 76, Ottawa, Canada, 1976.

9. Brainerd, Walter S. "The Minimization of Tree Automata." Information and Control 13 (1968): 484-491.

10. Brown, W. E. "Toward an Optimizing Compiler for a Very High Level Language." Ph.D. dissertation, Iowa State University, 1979.

11. Burgess, R. A. "Proof of Correctness of a Partial Parser for a Syntactically Ambiguous Language." Master's paper, Dept. of Computer Science, Iowa State University, 1974.

12. Compton, M. T. "APL in PL/I." Report RC 4481, IBM Research, Yorktown Heights, August, 1973.

13. Elshoff, J. "An Analysis of Some Commercial PL/I Programs." Software--Practice and Experience 6 (1976): 505-525.

14. Engelfriet, J. "Bottom-up and Top-down Tree Transformations-A Comparison." Math Systems Theory, 9 (1975): 198-231.

15. Falkoff, A. D. and Iverson, K. E. APLSV User's Manual. White Plains, N.Y.: IBM, 1973.

16. Falkoff, A. D. and Iverson, K. E. "The Design of APL." IBM J. Res. and Develop. 17 (July 1973): 324-334.

17. Geller, D. P. and Freedman, D. P. Structured Programming in APL. Cambridge, Massachusetts: Winthrop Publishers, Inc., 1976.

18. Guibas, L. J. and Wyatt, D. K. "Compilation and Delayed Evaluation in APL." Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona, 1978.

19. Hoffmann, C. M. "Matching Tree Patterns." Technical Report, Purdue University, West Lafayette, Indiana, 1978.

20. Hopcroft, J. E. "An n log n Algorithm for Minimizing States in a Finite State Automaton." In Theory of Machines and Computations, pp. 189-196. Edited by Z. Kohavi and A. Paz. New York: Academic Press, 1971.

21. Iverson, K. E. A Programming Language. New York: John Wiley and Sons, 1962.

22. Jenkins, M. A. "Translating APL, an Empirical Study." APL 75, Pisa, Italy, 1975.

23. Katzan, J., Jr. APL Programming and Computer Techniques. New York: Van Nostrand Reinhold, 1970.

24. Kleene, S. C. "Representation of Events in Nerve Nets and Finite Automata." In Automata Studies, pp. 3-41. Edited by C. Shannon and J. McCarthy. Princeton, N.J.: Princeton University Press, 1956.

25. Knuth, D. E. The Art of Programming, Vol. 1. Reading, Massachusetts: Addison-Wesley Publishing Co., 1968.

26. Knuth, D. E. "An Empirical Study of FORTRAN Programs." Software--Practice and Experience 1 (1971): 105-133.

27. Knuth, D., Morris, J. and Pratt, V. "Fast Pattern Matching in Strings." SIAM J. on Computing 6, No. 2 (1977): 323-350.

28. Lathwell, R. H. and Mezei, J. E. A Formal Description of APL. IBM Corporation Technical Report No. 320-3008, 1971.

29. McCormick, R. "Analysis of Storage Management Allocation." Master's paper, Dept. of Computer Science, Iowa State University, 1976.

30. Miller, T. C. "Tentative Compilation: A Design for an APL Compiler." Ph.D. dissertation, Yale University, 1978.

31. Miller, T. C. "Type Checking in an Imperfect World." Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona, 1978.

32. Morris, R. "An Investigation of Phrase Matching." Master's paper, Dept. of Computer Science, Iowa State University, 1980.

33. Omdahl, S. J. "Optimized Codes for Selected APL Idioms." Master's paper, Dept. of Computer Science, Iowa State University, in preparation.

34. Pakin, S. APL 360 Reference Manual. Second Edition. Chicago: Science Research Associates, 1972.

35. Perlis, A. J. "Steps Toward an APL Compiler - Updated." Research Report #24, Department of Computer Science, Yale University, 1975.

36. Perlis, A. J. and Rugaber, S. "The APL Idiom List." Research Report #87, Department of Computer Science, Yale University, April, 1977.

37. Polivka, R. P. and Pakin, S. APL: The Language and Its Usage. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975.

38. Preissman, S. "APL 360 Workspaces: Structure and Translation." Master's paper, Dept. of Computer Science, Iowa State University, 1976.

39. Robinson, S. K. and Torsun, I. S. "An Empirical Analysis of FORTRAN Programs." The Computer Journal 19, No. 1 (Feb. 1976): 56-62.

40. Roeder, R. "Type Determination in an Optimizing Compiler for APL." Ph.D. dissertation, Iowa State University, 1979.

41. Saal, H. J. and Weiss, Z. "An Empirical Study of APL Programs." Computer Languages 2, No. 3 (1977): 47-59.

42. Schwartz, J. T. "Optimization of Very High Level Languages - I: Value Transmission and Its Corollaries." Computer Languages 1, No. 2 (1975): 161-194.

43. Schwartz, J. T. "Optimization of Very High Level Languages - II: Deducing Relationships of Inclusion and Membership." Computer Languages 1, No. 3 (1975): 197-218.

44. Skewes, K. W. "Interpretation Versus Compilation: An Examination of the Resolution of Ambiguous Constructs in APL." Master's paper, Dept. of Computer Science, Iowa State University, 1975.

45. Stone, M. "A Non-recursive Phrase Match Algorithm." Master's paper, Dept. of Computer Science, Iowa State University, 1979.

46. Strawn, G. O. Heap Data, Virtual Memory, and APL. Dept. of Computer Science Technical Report #76-2, Iowa State University, 1975.

47. Strawn, G. O. "Does APL Really Need Run-time Parsing?" Software--Practice and Experience 7 (1977): 193-200.

48. Tafazzoli, Hassan B. "Phrase Matching for Tree Transducer." Master's paper, Dept. of Computer Science, Iowa State University, 1979.

49. Tarjan, R. E. and Yao, A. C. "Storing a Sparse Table." Comm. ACM 22, No. 11 (Nov. 1979): 606-611.

50. Thatcher, James W. "Tree Automata: An Informal Survey." In Currents in the Theory of Computing, pp. 143-172. Edited by A. Aho. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.

51. Thatcher, J. W. and Wright, J. B. "Generalized Automata Theory with an Application to a Decision Problem of Second-Order Logic." IBM Research RC 1713, 1966.

52. Van Dyke, E. "A Dynamic Incremental Compiler for an Interpretive Language." Hewlett Packard Journal, July 1977, 17-23.

53. Wulf, W. A. "Trends in the Design and Implementation of Programming Languages." Computer, 13 (January 1980): 14-24.

ACKNOWLEDGMENTS

I express my sincerest appreciation to Professor George O. Strawn for his many helpful suggestions and detailed guidance of this work. I also wish to express my thanks to Professor Clair G. Maple for his continued financial support of my program. A special acknowledgment is due Professor R. Krishnaswamy for his wonderful teaching and encouragement which helped through all of my studies. My sincere thanks to Professors Arthur V. Pohm and Terry A. Smay for the courses they taught. Also, a special thanks to Professor I-Ming Shen (now at the TamKang University, Taiwan, ROC) for the significant contribution he has made to my growth as a student.

I would also like to thank Dr. Jerome Niebaum for his hospitable assistance in the editing of the final copy.

APPENDIX: PL/I IMPLEMENTATION OF AN IDIOM MATCHING MACHINE

```
//B288BIN  JOB  I4983,FENG,MSGLEVEL=(1,1)
//S1  EXEC  PLIXCLG,REGION.GO=720K,TIME.GO=(1,30)
//PLI.SYSIN  DD  *
 *PROCESS MARGINS(2,72,1);
 IDICM_BIN: PROC OPTIONS(MAIN);

 /*************************************************************/
 /*                                                           */
 /* THIS PROGRAM HAS THREE MAJOR PROCEDURES :                 */
 /*                                                           */
 /* CONSTRUCTION:  IT CONSTRUCTS A DETERMINISTIC FINITE       */
 /*    TREE AUTOMATON (DFA) FROM A SET OF IDIOMS WHICH        */
 /*    ARE DESCRIBED BY SOME REGULAR TREE EXPRESSION.         */
 /*    THREE SUBPROGRAMS ARE NEEDED:                          */
 /*    (1) LABEL_PASS: ASSIGN A UNIQUE STATE FOR EACH         */
 /*          INDIVIDUAL SUBTREE                               */
 /*    (2) NONDET_PASS: ASSIGN TRANSITION FUNCTIONS FOR       */
 /*          EACH VARIABLE AND IDENTIFY THE NON-DETER-        */
 /*          MINISTIC TRANSITION FUNCTIONS                    */
 /*    (3) COMPLETION_PASS: APPLY THE CONCEPT OF "SUBSET      */
 /*          CONSTRUCTION" METHOD AND GET A DETERMINISTIC     */
 /*          BINARY TREE AUTOMATON;                           */
 /*                                                           */
 /* MINIMIZATION:  BY CONSTRUCTING A PREVIOUS STATE TABLE,    */
 /*    THIS PROCEDURE MINIMIZES THE NUMBER OF THE STATES      */
 /*    OF THE ABOVE DFA;                                      */
 /*                                                           */
 /* SELECTION:  IT ACCEPTS AN INPUT EXPRESSION TREE AND       */
 /*    RECOGNIZES ALL POSSIBLE IDIOMS ON IT.  IF THERE        */
 /*    ARE OVERLAPPED MATCHED IDIOMS, THEN THE ONE WITH       */
 /*    THE BEST BENEFIT WOULD BE SELECTED.                    */
 /*                                                           */
 /*************************************************************/

 DCL
 /* CONSTANTS */
     LEN_OF_SYMBOL              FIXED BIN(15) INIT(1),
     BLANK                      CHAR(LEN_OF_SYMBOL) INIT(' '),
     L_PARNTHE                  CHAR(LEN_OF_SYMBOL) INIT('('),
     R_PARNTHE                  CHAR(LEN_OF_SYMBOL) INIT(')'),
```

```
    VARIABLE                    CHAR(LEN_OF_SYMBOL) INIT('V'),
    END_OF_CLASS                CHAR(LEN_OF_SYMBOL) INIT(';'),
    END_OF_IDIOMS               CHAR(LEN_OF_SYMBOL) INIT('$'),
    UNION                       CHAR(LEN_OF_SYMBOL) INIT('|'),
    PRODUCT                     CHAR(LEN_OF_SYMBOL) INIT('#'),
    CLOSURE                     CHAR(LEN_OF_SYMBOL) INIT('*'),
    DEAD_STATE                  FIXED BIN(15) INIT(-1),
    NULL                        BUILTIN,
    ROW                         FIXED BIN(15) INIT(1),
    COL                         FIXED BIN(15) INIT(2),

    SIZE_OF_IDIOM               FIXED BIN(15),
    MAX_SIZE_OF_IDIOMS          FIXED BIN(15) INIT(100),
    MAX_#_OF_ALPHA              FIXED BIN(15) INIT(100),
    #_OF_ALPHA                  FIXED BIN(15),
    ALPHA(0:MAX_#_OF_ALPHA)     CHAR(LEN_OF_SYMBOL) INIT(' '),
    MAX_#_OF_IDIOMS             FIXED BIN(15) INIT(10),
    #_OF_TREES                  FIXED BIN(15),
    #_OF_IDIOMS                 FIXED BIN(15),
    MAX_#_OF_NEST               FIXED BIN(15) INIT(10),
    MAX_LEN_OF_INPUT            FIXED BIN(15) INIT(160),

/* NODE INFORMATION */
    B_NODE_INX                  FIXED BIN(15),
    1 B_NODE (0:MAX_SIZE_OF_IDIOMS),
      5 MARK                    FIXED BIN(15),
      5 LLINK                   FIXED BIN(15),
      5 RLINK                   FIXED BIN(15),
      5 STATE                   FIXED BIN(15),/*-1: FROM. V.*/
      5 ALPHA_INX               FIXED BIN(15),/*0: VARIABLE */
      5 INFO (0:MAX_#_OF_NEST) FIXED BIN(15),/*COL 0 = # */
      5 BENEFIT                 FIXED BIN(15),
      5 NON_MARK                BIT(1),          /*NON_OVERLP*/
      5 PATH                    BIT(MAX_LEN_OF_INPUT) VARYING,
      5 PATHLIST_PTR            POINTER,

/* TRANSITION FUNCTIONS */
    DFA (*,*,*)                 FIXED BIN(15) CONTROLLED,
    MINIMAL (*,*,*)             FIXED BIN(15) CONTROLLED,
    STATE_TAB (*,*)             FIXED BIN(15) CONTROLLED,
    STATEVEC (*)                FIXED BIN(15) CONTROLLED,
    STATE_NOW                   FIXED BIN(15),
    STATE_OLD                   FIXED BIN(15),

/* INDIVIDUAL TREES */
    STACK_INX                   FIXED BIN(15),
    1 STACK (MAX_#_OF_IDIOMS),
      5 FROM                    FIXED BIN(15),
      5 TO                      FIXED BIN(15),
      5 CLOSURE_PTR             POINTER,
```

```
    1 TREE  (MAX_#_OF_IDIOMS),
      5 ROOT                    FIXED BIN(15),
      5 BRO                     FIXED BIN(15),
    1 NODE                      BASED (P),
      5 LOC                     FIXED BIN(15),
      5 NEXT                    POINTER,
    I_TREE_INX                  FIXED BIN(15),
    1 I_TREE (0:MAX_SIZE_OF_IDIOMS),
      5 ALPHA_INX               FIXED BIN(15),
      5 LLINK                   FIXED BIN(15),
      5 RLINK                   FIXED BIN(15),
      5 B_INX                   FIXED BIN(15),
      5 PATH                    BIT(MAX_LEN_OF_INPUT) VARYING,

    POSTFIX_LIST (MAX_#_OF_IDIOMS) CHAR(MAX_LEN_OF_INPUT)
                                VARYING,
    NONFINAL_CLOSURE_SW (*)     BIT(1) CONTROLLED,

/* FINAL STATES */
    FINAL_SW (*)                BIT(1) CONTROLLED,
    FINAL_ID# (*)               FIXED BIN(15) CONTROLLED,
    FINAL_NFAPTR (*)            POINTER CONTROLLED,
    FINAL_BENEFIT (*)           FIXED BIN(15) CONTROLLED,
    1 FINAL_PATH_INFO (*)       CONTROLLED,
      5 VAR_PTR                 POINTER,
      5 CLOSURE_PTR             POINTER,
      5 PRODUCT_PTR             POINTER,
    1 PATH_NODE                 BASED (P),
      5 P_INX                   FIXED BIN(15),
      5 P_PATH_LEN              FIXED BIN(15),
      5 P_PATH                  BIT(160),
      5 P_NEXT                  POINTER,
    1 PATH_NODE1                BASED (P),
      5 P_PATH_LEN1             FIXED BIN(15),
      5 P_PATH1                 BIT(160),
      5 P_NEXT1                 POINTER,

    MATCH_PHASE                 BIT(1),
    END_ONE_IDIOM               BIT(1);


 DO WHILE ('1'B);

    CALL CONSTRUCTION;

    CALL MINIMIZATION;

    CALL IDIOM_MATCHING;

 END;
```

```
/*************************************************************/
/*                                                           */
/*  CONSTRUCT A DETERMINISTIC BINARY TREE AUTOMATON          */
/*                                                           */
/*************************************************************/
CONSTRUCTION: PROC;

    CALL INDIVIDUAL_TREES;

    CALL REGULAR_PARSER;

    CALL REGULAR_POSTFIX;

    CALL NONDET_PASS;

    CALL COMPLETION_PASS;

    CALL DFA_FINAL;

    CALL PRINT_DFA;

    RETURN;
   END;

/*************************************************************/
/*                                                           */
/*             GET INPUT SYMBOL                              */
/*                                                           */
/*************************************************************/
GET_NEXT: PROC RETURNS (CHAR(1));

DCL X                    CHAR(1);

  GET EDIT (X) (A(1));
  IF X ¬= BLANK THEN DO;
                     PUT EDIT (X) (A(1));
                     RETURN (X);
                     END;
  IF END_ONE_IDIOM |
     MATCH_PHASE THEN    DO;
                         X = BLANK;
                         END_ONE_IDIOM = '0'B;
                         END;
  ELSE DO;
       X = VARIABLE;
       END_ONE_IDIOM = '1'B;
       END;

  RETURN (X);
 END GET_NEXT;
```

```
/*************************************************************/
/*                                                           */
/*              PRINT DFA                                    */
/*                                                           */
/*************************************************************/

PRINT_DFA: PROC;

DCL (I, J, K, L)            FIXED BIN(15);

 PUT SKIP(3) LIST('  THE DETERMINISTIC TREE AUTOMATON IS');

 DO I = 1 TO #_OF_ALPHA;
    PUT SKIP(2) EDIT(ALPHA(I)) (X(2),A);
    PUT SKIP EDIT ('      ') (A);
    DO L = 0 TO STATE_NOW;
       PUT EDIT ('(',L,')') (A,F(2),A);
    END;
    DO J = 0 TO STATE_NOW;
       PUT SKIP EDIT('(',J,')') (A,F(2),A);
       DO K = 0 TO STATE_NOW;
          PUT EDIT(DFA(I,J,K)) (F(4));
       END;
    END;
 END;

 PUT SKIP(3) LIST('  THE STATES IN DFA ARE');  PUT SKIP;
 DO I = 0 TO STATE_NOW;
    PUT SKIP EDIT('<',I,'>---') (A,F(4),A);
    PUT EDIT('{ ') (A);
    DO J = 1 TO STATE_TAB(I,0)-1;
       PUT EDIT('(',STATE_TAB(I,J),'),') (A,F(2),A);
    END;
 PUT EDIT('(',STATE_TAB(I,STATE_TAB(I,0)),') }') (A,F(2),A);
 END;

 PUT SKIP(3) EDIT('  THE FINAL STATES ARE ') (A);
 DO I = 0 TO STATE_NOW;
    IF FINAL_SW (I) THEN PUT EDIT ('(',I,') ') (A,F(2),A);
 END;

 RETURN;

 END PRINT_DFA;
```

```
/*************************************************************/
/*************************************************************/
/*                                                           */
/*        INPUT INDIVIDUAL TREES                             */
/*                                                           */
/*************************************************************/
/*************************************************************/
INDIVIDUAL_TREES: PROC;
DCL
    X                          CHAR(LEN_OF_SYMBOL);

  ON ENDFILE (SYSIN)
      STOP;

GET SKIP EDIT (X) (A(1));
PUT PAGE LIST('  INPUT INDIVIDUAL TREES ARE'); PUT SKIP;
B_NODE_INX = 0;

I_TREE_INX = 0;     I_TREE = 0;
#_OF_ALPHA = 0;      ALPHA(0) = VARIABLE;

END_ONE_IDIOM = '0'B; #_OF_TREES = 0; MATCH_PHASE = '0'B;
DO WHILE (X¬=END_OF_IDIOMS);
   #_OF_TREES = #_OF_TREES + 1;
   PUT SKIP EDIT (#_OF_TREES,'-  ',X) (F(2),A,A);
   TREE(#_OF_TREES).ROOT = NARY_TO_BIN ('',X);
   TREE(#_OF_TREES).BRO = I_TREE_INX;
   GET SKIP EDIT (X) (A(1));
END;

SIZE_OF_IDIOM = I_TREE_INX;

RETURN;

END INDIVIDUAL_TREES;
```

```
/*****************************************************************/
/*                                                               */
/*     GET BINARY REPRESENTATION OF EACH INDIVIDUAL TREE         */
/*                                                               */
/*****************************************************************/

NARY_TO_BIN: PROC(PATH,X) RECURSIVE RETURNS(FIXED BIN(15));

DCL

    (P,I)                          FIXED BIN(15),
    PATH                           BIT(*) VARYING,
    FIND                           BIT(1),
    X                              CHAR(*);

   IF X = R_PARNTHE | X = BLANK THEN
      RETURN (0);

   I_TREE_INX = I_TREE_INX + 1;
   P = I_TREE_INX;
   I_TREE(P).PATH = PATH;

   FIND = '0'B;
   DO I = 0 TO #_OF_ALPHA WHILE (¬FIND);
      IF X = ALPHA(I) THEN DO;
                           I_TREE(P).ALPHA_INX = I;
                           FIND = '1'B;
                           END;
   END;
   IF ¬FIND THEN DO;                  /* INSERT A NEW SYMBOL */
                 #_OF_ALPHA = #_OF_ALPHA + 1;
                 ALPHA (#_OF_ALPHA) = X;
                 I_TREE(P).ALPHA_INX = #_OF_ALPHA;
                 END;

   X = GET_NEXT;

   IF X = L_PARNTHE THEN DO;
               X = GET_NEXT;
               I_TREE(P).LLINK = NARY_TO_BIN (PATH||'0'B,X);
               X = GET_NEXT;
               END;
   ELSE I_TREE(P).LLINK = 0;

   I_TREE(P).RLINK = NARY_TO_BIN (PATH||'1'B,X);

   RETURN (P);

END NARY_TO_BIN;
```

```
/*****************************************************/
/*****************************************************/
/*                                                   */
/*    REGULAR EXPRESSIONS ARE PARSED BY A LL(1) PARSER   */
/*                                                   */
/*****************************************************/
/*****************************************************/
REGULAR_PARSER: PROC;

DCL
    (OPERATOR, SYMBOL)              CHAR(1),
    I                               FIXED BIN(15);
            /* TEE INPUT REGULAR EXPRESSIONS ARE DEFINED */
            /* BY A GRAMMAR <EXP> WITH THE CLOSURE OPER- */
            /* ATOR HAS THE HIGHEST PRECEDENCE, THEN     */
            /* PRODUCT, THEN UNION.                      */

  PUT SKIP (2) EDIT('  INPUT IDIOMS ARE') (A);  PUT SKIP;
  GET SKIP EDIT(OPERATOR,SYMBOL) (A(1),A(1));
  I = 0;   POSTFIX_LIST = '';

  DO WHILE(OPERATOR ¬= END_OF_IDIOMS);
     I = I + 1;
     PUT SKIP EDIT (I,'. ',OPERATOR,SYMBOL) (F(2),A,A,A);
     CALL EXP;
     GET SKIP EDIT(OPERATOR,SYMBOL) (A(1),A(1));
  END;

  #_OF_IDIOMS = I;

  RETURN;
/*****************************************************/
/*      <EXP> ---> <ITEM> <EXP_LIST>                 */
/*****************************************************/

EXP:  PROC RECURSIVE;

      CALL ITEM;

      CALL EXP_LIST;

      RETURN;
  END EXP;
```

```
/*************************************************************/
/*      <EXP_LIST> ---> UNION <ITEM> <EXP_LIST>              */
/*                 ---> EPSILON                              */
/*************************************************************/
EXP_LIST: PROC RECURSIVE;

DCL   X                          CHAR(LEN_OF_SYMBOL);

      IF OPERATOR = UNION THEN
         DO;
         X = SYMBOL;
         GET EDIT(OPERATOR,SYMBOL) (A(1),A(1));
         PUT EDIT(' ',OPERATOR,SYMBOL) (A,A,A);
         CALL ITEM;
         POSTFIX_LIST(I) = POSTFIX_LIST(I) || UNION || X;
         CALL EXP_LIST;
         END;
      RETURN;
  END EXP_LIST;

/*************************************************************/
/*      <ITEM> ---> <UNIT> <ITEM_LIST>                       */
/*************************************************************/
ITEM: PROC RECURSIVE;

      CALL UNIT;
      CALL ITEM_LIST;
      RETURN;
  END ITEM;

/*************************************************************/
/*      <ITEM_LIST> ---> PRODUCT <UNIT> <ITEM_LIST>          */
/*                  ---> EPSILON                             */
/*************************************************************/
ITEM_LIST: PROC RECURSIVE;

DCL X                              CHAR(LEN_OF_SYMBOL);

      IF OPERATOR = PRODUCT THEN
         DO;
         X = SYMBOL;
         GET EDIT (OPERATOR,SYMBOL) (A(1),A(1));
         PUT EDIT (' ',OPERATOR,SYMBOL) (A,A,A);
         CALL UNIT;
         POSTFIX_LIST(I) = POSTFIX_LIST(I) || PRODUCT || X;
         CALL ITEM_LIST;
         END;
      RETURN;
 END ITEM_LIST;
```

```
/*************************************************************/
/*       <UNIT> ---> <B_TREE> CLOSURE                        */
/*              ---> <B_TREE>                                */
/*************************************************************/
UNIT: PROC RECURSIVE;

      CALL B_TREE;

      IF OPERATOR = CLOSURE THEN
         DO;
         POSTFIX_LIST(I)=POSTFIX_LIST(I)||CLOSURE||SYMBOL;
         GET EDIT (OPERATOR,SYMBOL) (A(1),A(1));
         PUT EDIT (' ',OPERATOR,SYMBOL) (A,A,A);
         END;

      RETURN;
  END UNIT;

/*************************************************************/
/*    <B_TREE> ---> LEFT_PARNTHSIS <EXP> RIGHT_PARNTHSIS     */
/*              ---> INTEGER                                 */
/*************************************************************/
B_TREE: PROC RECURSIVE;

      IF OPERATOR = L_PARNTHE THEN
         DO;
         GET EDIT (OPERATOR,SYMBOL) (A(1),A(1));
         PUT EDIT (' ',OPERATOR,SYMBOL) (A,A,A);
         CALL EXP;
         END;

      ELSE POSTFIX_LIST(I) = POSTFIX_LIST(I) ||
                             OPERATOR || SYMBOL;

      GET EDIT (OPERATOR,SYMBOL) (A(1),A(1));
      PUT EDIT (' ',OPERATOR,SYMBOL) (A,A,A);

      RETURN;
  END B_TREE;

 END REGULAR_PARSER;
```

```
/*************************************************************/
/*        EVALUATE THE POSTFIX REGULAR EXPRESSIONS           */
/*                                                           */
/*  EACH IDIOM IS REPRESENTED BY A REGULAR EXPRESSION IN     */
/*  POSTFIX ORDER.  EACH INDIVIDUAL TREE HAS BEEN INPUT      */
/*  IN THE SAME ORDER.  ONE FINAL STATE FOR EACH RECORD      */
/*************************************************************/
REGULAR_POSTFIX: PROC;
DCL DIGITS                  CHAR(9) INIT('123456789'),
    (I, #, J, R, B, CURR)   FIXED BIN(15),
    (OPERATOR, SYMBOL)      CHAR(1);

  STACK_INX = 0;       STATE_NOW = 0;          B_NODE_INX = 0;
  B_NODE(0).ALPHA_INX=-1; /* DUMMY NODE IS A NON_VAR NODE*/
  ALLOCATE FINAL_PATH_INFO(0:MAX_SIZE_OF_IDIOMS);
  ALLOCATE NONFINAL_CLOSURE_SW (0:MAX_SIZE_OF_IDIOMS);
     NONFINAL_CLOSURE_SW = '0'B;  FINAL_PATH_INFO = NULL;
  ALLOCATE FINAL_BENEFIT (0:MAX_SIZE_OF_IDIOMS) INIT (0);
  ALLOCATE FINAL_ID# (0:MAX_SIZE_OF_IDIOMS) INIT(0);

  DO # = 1 TO #_OF_IDIOMS;
   DO J = 1 TO LENGTH(POSTFIX_LIST(#)) BY 2;
    OPERATOR = SUBSTR(POSTFIX_LIST(#),J,1);
    SYMBOL = SUBSTR(POSTFIX_LIST(#),J+1,1);
    SELECT (OPERATOR);
      WHEN (UNION)    CALL UNION_DFA;
      WHEN (PRODUCT) CALL PRODUCT_DFA;
      WHEN (CLOSURE) CALL CLOSURE_DFA;
      OTHERWISE DO;       /* OPERAND, TREE */
       STACK_INX = STACK_INX + 1;
       I = INDEX(DIGITS,OPERATOR)*10+INDEX(DIGITS,SYMBOL);
       B_NODE_INX=B_NODE_INX+(TREE(I).BRO-TREE(I).ROOT+1);
       CALL LABEL_PASS(TREE(I).ROOT,TREE(I).BRO);
       R, STACK(STACK_INX).FROM=I_TREE(TREE(I).ROOT).B_INX;
       B, STACK(STACK_INX).TO = I_TREE(TREE(I).BRO).B_INX;
       STACK(STACK_INX).CLOSURE_PTR = NULL;
       FINAL_BENEFIT(B_NODE(R).STATE) =
          TREE(I).BRO - TREE(I).ROOT;
       DO CURR = B-1 TO R BY -1;              /* TREE||V IN */
          IF B_NODE(CURR).ALPHA_INX = 0 &  /* BIN REP    */
             B_NODE(R).STATE ¬= -1 THEN    /* TREE = V   */
             CALL ADD_PATH(1,CURR,B_NODE(R).STATE);
        END;
       NONFINAL_CLOSURE_SW (B_NODE(R).STATE) = '1'B;
       FINAL_ID# (B_NODE(R).STATE) = #;
       END;
    END;
   END;
  END;
  RETURN;
```

```
/*************************************************************/
/*       UNION                                               */
/*************************************************************/
UNION_DFA: PROC;

  DCL    (R_BRO,R_ROOT,CURR)          FIXED BIN(15),
         R_PTR                        POINTER;

  R_ROOT = STACK(STACK_INX).FROM;
  R_BRO = STACK(STACK_INX).TO;
  R_PTR = STACK(STACK_INX).CLOSURE_PTR;

  STACK_INX = STACK_INX - 1;                  /* POP          */

  STACK(STACK_INX).TO = R_BRO;                /* POP & PUSH   */
  CURR = STACK(STACK_INX).FROM;
  B_NODE(CURR).INFO(0) = B_NODE(CURR).INFO(0) + 1;
  B_NODE(CURR).INFO(B_NODE(CURR).INFO(0)) =
                B_NODE(R_ROOT).STATE;
  CALL FILL_INFO (CURR,R_ROOT);
  CALL CLOSURE_LIST (STACK_INX,R_PTR);

  RETURN;
 END UNION_DFA;

/*************************************************************/
/*       PRODUCT                                             */
/*************************************************************/
PRODUCT_DFA: PROC;

  DCL
      (R_PTR, Q)                      POINTER,
      (R_ROOT, R_BRO, CURR, #, X)     FIXED BIN(15);

  R_ROOT = STACK(STACK_INX).FROM;
  R_BRO = STACK(STACK_INX).TO;
  R_PTR = STACK(STACK_INX).CLOSURE_PTR;

  STACK_INX = STACK_INX - 1;                  /* POP          */

DO CURR=STACK(STACK_INX).TO TO STACK(STACK_FROM) BY -1;
     IF B_NODE(CURR).LLINK = 0 &              /*FRONTIER NODE*/
        B_NODE(CURR).MARK=0 &
        SYMBOL = ALPHA(B_NODE(CURR).ALPHA_INX)  THEN
           DO;
           B_NODE(CURR).STATE = B_NODE(R_ROOT).STATE;
           IF R_PTR ¬= NULL THEN DO;  /*INCLUDE THE STATE*/
              Q = R_PTR;              /*OF CLOSURE FRONT.*/
              DO WHILE (Q¬=NULL);
               #,B_NODE(CURR).INFO(0) =
```

```
                        B_NODE(CURR).INFO(0) + 1;
                    B_NODE(CURR).INFO(#)=B_NODE(Q->LOC).STATE;
                    Q = Q->NEXT;
                    END;
                   END;
               B_NODE(CURR).MARK = 1;
               CALL FILL_INFO (CURR,B_ROOT);
               X=GET_ROOT(CURR,STACK(STACK_INX).TO,
                          STACK(STACK_INX).FROM);
               CALL ADD_PATH(2,CURR,B_NODE(X).STATE);
               END;
 END;

 STACK(STACK_INX).TO = R_BRO;              /* POP & PUSH   */

 RETURN;

GET_ROOT: PROC (CURR,I,J) RETURNS (FIXED BIN(15));
DCL  (CURR, I, J, K, L, M, X, LEN)  FIXED BIN(15),
     (FIND, ERROR)                  BIT(1);
FIND = '0'B;       X = 0;
LEN = LENGTH (B_NODE(CURR).PATH);
DO K = I TO J BY -1 WHILE (¬FIND);
   L = K;  ERROR = '0'B;
   DO M = 1 TO LEN WHILE (¬ERROR);
      IF SUBSTR(B_NODE(CURR).PATH,M,1) THEN
            L = B_NODE(L).RLINK;
      ELSE L = B_NODE(L).LLINK;
      IF L = 0 THEN ERROR = '1'B;
   END;
   IF CURR = L THEN DO; FIND = '1'B; X = K; END;
END;
RETURN (X);
END GET_ROOT;
END PRODUCT_DFA;

/*************************************************************/
/*       CLOSURE                                             */
/*************************************************************/
CLOSURE_DFA: PROC;

DCL   (CURR, S, R_STATE)      FIXED BIN(15),
      (P, Q)                  POINTER;
   S = 0;
   R_STATE = B_NODE(STACK(STACK_INX).FROM).STATE;
DO CURR=STACK(STACK_INX).TO TO STACK(STACK_INX).FROM BY -1;
      IF B_NODE(CURR).LLINK = 0 &          /*FRONTIER NODE*/
         SYMBOL = ALPHA(B_NODE(CURR).ALPHA_INX)  THEN
           DO;
           B_NODE(CURR).INFO(0) = B_NODE(CURR).INFO(0)+1;
```

```
            B_NODE(CURR).INFO(B_NODE(CURR).INFO(0))=R_STATE;
            S = CURR;
            CALL ADD_PATH(3,CURR,R_STATE);
            END;
   END;

   IF S ¬= 0 THEN DO;
                  ALLOCATE NODE SET (P);
                  Q = STACK(STACK_INX).CLOSURE_PTR;
                  STACK(STACK_INX).CLOSURE_PTR = P;
                  P->LOC = S;        /* INX TO B_NODE VEC */
                  P->NEXT = Q;
                  END;

   RETURN;
  END CLOSURE_DFA;

/*****************************************************************/
/*     FILL INFO FIELD WITH THE STATES OF RIGHT OPERAND          */
/*****************************************************************/

FILL_INFO: PROC (CURR,R_ROOT);

DCL  (I, J, CURR, R_ROOT)     FIXED BIN(15);

  I = B_NODE(CURR).INFO(0);
  DO J = 1 TO B_NODE(R_ROOT).INFO(0);
     B_NODE(CURR).INFO(I+J) = B_NODE(R_ROOT).INFO(J);
  END;
  B_NODE(CURR).INFO(0) = I + B_NODE(R_ROOT).INFO(0);
  RETURN;
 END FILL_INFO;

/*****************************************************************/
/*     INSERT R_PTR AT FRONT OF THE CLOSURE LIST                 */
/*****************************************************************/

CLOSURE_LIST: PROC (STACK_INX,R_PTR);

 DCL    (P,R_PTR)            POINTER,
        STACK_INX            FIXED BIN(15);

   IF R_PTR ¬= NULL THEN
      DO;
      P = STACK(STACK_INX).CLOSURE_PTR;
      IF P = NULL THEN STACK(STACK_INX).CLOSURE_PTR=R_PTR;
      ELSE DO;
           DO WHILE (P->NEXT¬=NULL);
              P = P->NEXT;
            END;
```

```
            P->NEXT = R_PTR;
            END;
        END;
    RETURN;
  END CLOSURE_LIST;

/*************************************************************/
/*      INSERT PATH NODE AT FRONT OF FINAL_PATH_INFO         */
/*************************************************************/
ADD_PATH: PROC (TYPE,CURR,R_STATE);

 DCL   (TYPE, CURR, R_STATE)                FIXED BIN(15),
       P                                    POINTER;

  ALLOCATE PATH_NODE SET (P);
  P->P_INX = B_NODE(CURR).ALPHA_INX;
  P->P_PATH = B_NODE(CURR).PATH;
  P->P_PATH_LEN = LENGTH(B_NODE(CURR).PATH);
  IF TYPE = 1 THEN DO;
       P->P_NEXT = FINAL_PATH_INFO(R_STATE).VAR_PTR;
       FINAL_PATH_INFO(R_STATE).VAR_PTR = P;
       END;
  ELSE IF TYPE = 2 THEN DO;
       P->P_NEXT = FINAL_PATH_INFO(R_STATE).PRODUCT_PTR;
       FINAL_PATH_INFO(R_STATE).PRODUCT_PTR = P;
       END;
  ELSE DO;
       P->P_NEXT = FINAL_PATH_INFO(R_STATE).CLOSURE_PTR;
       FINAL_PATH_INFO(R_STATE).CLOSURE_PTR = P;
       END;

  RETURN;
END ADD_PATH;

END REGULAR_POSTFIX;
```

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*  ASSIGN THE SAME STATE # TO ALL COMMON SUBEXPRESSIONS         */
/*                                                               */
/*****************************************************************/
/*****************************************************************/

LABEL_PASS: PROC (LOW,UP);

DCL (CURR, L, R, I, J, K, INX, LOW, UP)    FIXED BIN(15);

 INX = B_NODE_INX;
 DO CURR = UP TO LOW BY -1;       /* REVERSE OF PREORDER   */

    I, B_NODE(INX).ALPHA_INX = I_TREE(CURR).ALPHA_INX;
    L, B_NODE(INX).LLINK=I_TREE(I_TREE(CURR).LLINK).B_INX;
    R, B_NODE(INX).RLINK=I_TREE(I_TREE(CURR).RLINK).B_INX;
    B_NODE(INX).PATH = I_TREE(CURR).PATH;
    I_TREE(CURR).B_INX = INX;

    IF I = 0 & R = 0          /* FRONTIER VAR          */
       | I = 0 & B_NODE(R).STATE = -1
       THEN B_NODE(INX).STATE = -1;/* VAR NODE WITH RIGHT*/
                                   /* FRONTIER VAR SON   */
    ELSE DO;
         STATE_NOW = STATE_NOW + 1;
         B_NODE(INX).STATE = STATE_NOW;
         END;
    B_NODE(INX).MARK, B_NODE(INX).BENEFIT = 0;
    B_NODE(INX).INFO (*) = 0;
    B_NODE(INX).NON_MARK = '0'B;
    B_NODE(INX).PATHLIST_PTR = NULL;

    INX = INX - 1;
 END;
 RETURN;

 END LABEL_PASS;
```

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*     GET TRANSITION FUNCTION VALUES FOR INDIVIDUAL TREES       */
/*                                                               */
/*****************************************************************/
/*****************************************************************/

NONDET_PASS: PROC;

DCL STATE_LIST (*,*)          FIXED BIN(15) CONTROLLED;
                  /* COL 0 SAVES THE # OF STATES          */
                  /* ROW 1 = LIST OF STATES OF LEFT SON   */
                  /* ROW 2 = LIST OF STATES OF RIGHT SON  */

DCL (T, L, R, S, I, J, K, JJ, KK, CURR)     FIXED BIN(15);

                  /* FILL UP ALL TRANSITION FUNCTIONS     */
                  /* RELATED TO THE VARIABLE NODES        */

 STATE_OLD = STATE_NOW;

 ALLOCATE DFA(0:#_OF_ALPHA,-1:STATE_OLD*3,-1:STATE_OLD*3);
         DFA = 0;                        /* DFA(0,*,*) IS FOR  */
                                         /* VARIABLE           */
 ALLOCATE STATE_TAB(0:STATE_OLD*3,0:STATE_OLD);
 ALLOCATE STATE_LIST(2,0:STATE_OLD+1);
 STATE_LIST = 0;          STATE_TAB = 0;

 DO I = 0 TO STATE_OLD;
    STATE_TAB(I,0) = 1;
    STATE_TAB(I,1) = I;
 END;
                                        /* REVERSE OF PREORDER*/

 DO CURR = B_NODE_INX TO 1 BY -1;

    S = B_NODE(CURR).STATE;

       T = B_NODE(CURR).ALPHA_INX;
       L = B_NODE(CURR).LLINK;
       R = B_NODE(CURR).RLINK;

       IF S = -1 & R = 0              /* SKIP A FRONTIER VAR*/
         | S = -1 & B_NODE(R).STATE = -1 THEN ;
       ELSE IF T = 0 & B_NODE(R).STATE = -1 THEN
               B_NODE(CURR).STATE = -1;

       ELSE IF B_NODE(CURR).MARK=0 THEN
               DO;
```

```
                IF T = 0 THEN L = -1;/* LEFT STATE LIST    */
                CALL GET_STATE_LIST(L,ROW);
                CALL GET_STATE_LIST(R,COL); /* RIGHT       */

                DO J = 1 TO STATE_LIST(ROW,0);
                   JJ = STATE_LIST(ROW,J);
                   DO K = 1 TO STATE_LIST(COL,0);
                      KK = STATE_LIST(COL,K);
                      IF T = 0 THEN
                           DO I = 1 TO #_OF_ALPHA;
                              IF DFA(I,JJ,KK) ¬= 0 THEN
                                 DFA(I,JJ,KK) =
                                 NEW_STATE(DFA(I,JJ,KK),S);
                              ELSE DFA(I,JJ,KK) = S;
                           END;
                      ELSE IF DFA(T,JJ,KK) ¬= 0 THEN
                              DFA(T,JJ,KK) =
                                NEW_STATE(DFA(T,JJ,KK),S);
                           ELSE DFA(T,JJ,KK) = S;
                   END;
                  END;  /* OF J LOOP */
                END;
 END;    /* OF DO LOOP */

 RETURN;

/*************************************************************/
/*                                                           */
/*             GET STATE LIST FOR L OR R SONS                */
/*                                                           */
/*************************************************************/

GET_STATE_LIST: PROC (LR,I);

DCL
   (LR, I, J, STATE)                    FIXED BIN(15);

  IF LR = -1 THEN STATE = -1;       /* CURR NODE IS A VAR */
  ELSE STATE = B_NODE(LR).STATE;

  IF STATE = -1 THEN
       CALL FILL_ALL;

  ELSE DO;
       STATE_LIST(I,0) = B_NODE(LR).INFO(0) + 1;
       STATE_LIST(I,1) = STATE;
       DO J = 1 TO B_NODE(LR).INFO(0);
          IF B_NODE(LR).INFO(J) = -1 THEN DO;
             CALL FILL_ALL;
             RETURN;
```

```
            END;
          STATE_LIST(I,J+1) = B_NODE(LR).INFO(J);
         END;
       END;

  RETURN;

  FILL_ALL: PROC;
     STATE_LIST(I,0) = STATE_OLD + 1;
     DO J = 0 TO STATE_OLD;
        STATE_LIST(I,J+1) = J;
      END;
     RETURN;
     END FILL_ALL;

 END GET_STATE_LIST;

/*****************************************************************/
/*                                                               */
/*              GET A NEW STATE                                  */
/*                                                               */
/*****************************************************************/
NEW_STATE: PROC (OLD,NEW) RETURNS(FIXED BIN(15));

DCL     (I, K, #, OLD, NEW)        FIXED BIN(15),
        (INSERT, FIND)             BIT(1);

  IF OLD = NEW THEN RETURN(OLD);

  STATE_NOW = STATE_NOW + 1;
  STATE_TAB(STATE_NOW,0) = STATE_TAB(OLD,0) + 1;
  K = 0;      I = 1;
  INSERT = '0'B;

  DO WHILE (I <= STATE_TAB(OLD,0) & ¬INSERT);
     IF STATE_TAB(OLD,I) = NEW THEN
        DO;
        STATE_NOW = STATE_NOW - 1;
        RETURN(OLD);
        END;
     ELSE IF STATE_TAB(OLD,I) < NEW THEN
        DO;
        K = K + 1;
        STATE_TAB(STATE_NOW,K) = STATE_TAB(OLD,I);
        I = I + 1;
        END;
     ELSE DO;
          K = K + 1;
          STATE_TAB(STATE_NOW,K) = NEW;
          INSERT = '1'B;
```

```
         END;
   END;

   IF INSERT THEN
      DO WHILE (I <= STATE_TAB(OLD,0));
         K = K + 1;
         STATE_TAB(STATE_NOW,K) = STATE_TAB(OLD,I);
         I = I + 1;
      END;
   ELSE DO;
        K = K + 1;
        STATE_TAB(STATE_NOW,K) = NEW;
        END;

                          /* CHECK IF ROW(STATE_NOW) EXIST */
   # = STATE_TAB(STATE_NOW,0);
   FIND = '0'B;
   DO I = 1 TO STATE_NOW-1 WHILE (¬FIND);
      IF # = STATE_TAB(I,0) THEN
         DO;
         FIND = '1'B;
         DO K = 1 TO # WHILE (FIND);
            IF STATE_TAB(STATE_NOW,K) ¬= STATE_TAB(I,K) THEN
               FIND = '0'B;
           END;
         IF FIND THEN DO;
                      STATE_NOW = STATE_NOW - 1;
                      RETURN (I);
                      END;
         END;
   END;    /* OF I LOOP */
   RETURN (STATE_NOW);

END NEW_STATE;

END NONDET_PASS;
```

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*   FIND ALL NEW STATES AND THEIR TRAN. FUNCTION VALUES         */
/*                                                               */
/*****************************************************************/
/*****************************************************************/
COMPLETION_PASS: PROC;

 DCL
   (I, J, K, T)                          FIXED BIN(15),
   T_VEC (*)                             BIT(1) CONTROLLED;

 ALLOCATE T_VEC (1:STATE_OLD);

 T = STATE_NOW + 1;              /* INX TO NEW STATE */
 J = STATE_OLD + 1;

 DO WHILE (STATE_NOW >= J);
    DO I = 1 TO #_OF_ALPHA;

       DO K = 0 TO J-1;

          CALL DFA_VALUE(I,J,K,ROW);
       END;

       DO K = 0 TO J;

          CALL DFA_VALUE(I,K,J,COL);
       END;

    END;      /* OF I LOOP */

    J = J + 1;

 END;

 RETURN;
```

```
/*************************************************************/
/*                                                           */
/*              GET DFA VALUE                                */
/*                                                           */
/*************************************************************/
DFA_VALUE: PROC(I,J,K,R_C);

DCL
   (I, J, K, R_C, #, #_S, STATE_INX, T_INX, JJ, KK)
                                 FIXED BIN(15),
   (FIND,EQUAL)                  BIT(1);

 IF R_C = ROW THEN #_S = STATE_TAB(J,0);
 ELSE #_S = STATE_TAB(K,0);

 T_VEC = '0'B;

 DO STATE_INX = 1 TO #_S;
    IF R_C = ROW THEN # = DFA(I,STATE_TAB(J,STATE_INX),K);
    ELSE # = DFA(I,J,STATE_TAB(K,STATE_INX));

    IF # = 0 THEN ;

    ELSE IF # <= STATE_OLD THEN
           T_VEC(#) = '1'B;

    ELSE DO #_S = 1 TO STATE_TAB(#,0);
           T_VEC(STATE_TAB(#,#_S)) = '1'B;
         END;
 END;

 STATE_TAB(T,0) = 0;

 DO T_INX = 1 TO STATE_OLD;
    IF T_VEC(T_INX) THEN
       DO;
       STATE_TAB(T,0) = STATE_TAB(T,0) + 1;
       STATE_TAB(T,STATE_TAB(T,0)) = T_INX;
       END;
 END;

 # = STATE_TAB(T,0);

 IF # = 0 THEN
    DFA(I,J,K) = 0;

 ELSE IF # = 1 THEN
    DFA(I,J,K) = STATE_TAB(T,1);

 ELSE DO;
```

```
      FIND = '0'B;
      DO JJ = STATE_OLD+1 TO STATE_NOW WHILE (¬FIND);
         IF # = STATE_TAB(JJ,0) THEN
            DO;
            EQUAL = '1'B;
            DO KK = 1 TO # WHILE (EQUAL);
               IF STATE_TAB(T,KK) ¬= STATE_TAB(JJ,KK) THEN
                  EQUAL = '0'B;
              END;
            IF EQUAL THEN
               DO;
               DFA(I,J,K) = JJ;
               FIND = '1'B;
               END;
            END;

      END;  /* OF JJ LOOP */

      IF ¬FIND THEN          /* A NEW STATE */
         DO;
         STATE_NOW = T;
         DFA(I,J,K) = T;
         T = STATE_NOW + 1;
         END;

      END;

RETURN;

END DFA_VALUE;

END COMPLETION_PASS;
```

```
/*************************************************************/
/*                                                           */
/*            GET THE FINAL STATES OF DFA                    */
/*                                                           */
/*************************************************************/
DFA_FINAL: PROC;

 DCL  I                          FIXED BIN(15),
      P                          POINTER;

  ALLOCATE FINAL_SW (0:STATE_NOW) INIT('0'B);

  DO I = 1 TO STACK_INX;
     CALL UP_SW (I,STACK(I).FROM);

     P = STACK(I).CLOSURE_PTR;
     DO WHILE (P¬=NULL);
        CALL UP_SW (I,P->LOC);
        P = P->NEXT;
     END;
  END;

  RETURN;

 UP_SW: PROC (I,R);

 DCL   (I, J, S, R)              FIXED BIN(15);

  S = B_NODE(R).STATE;
  FINAL_SW (S) = '1'B;       FINAL_ID# (S) = I;
  DO J = 1 TO B_NODE(R).INFO(0);
     S = B_NODE(R).INFO(J);
     FINAL_SW (S) = '1'B;     FINAL_ID# (S) = I;
  END;

  RETURN;
 END UP_SW;

END DFA_FINAL;
```

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*            MINIMIZATION                                       */
/*                                                               */
/*****************************************************************/
/*****************************************************************/
MINIMIZATION: PROC;

 DCL

     BLOCKVEC (*)                    CTL FIXED BIN(15),
     BLOCKSIZE (*)                   CTL FIXED BIN(15),
     #_OF_BLOCKS                     FIXED BIN(15),

     P_TAB (*,*)                     CONTROLLED POINTER,
     1 PAIR                          BASED (P),
       5 ROW#                        FIXED BIN(15),
       5 COL#                        FIXED BIN(15),
       5 PTR                         POINTER,

     L (*,*)                         CONTROLLED BIT(1),

     BLK_LIST (*,*)                  CONTROLLED BIT(1),
     BLK_HDR (*,*)                   CONTROLLED BIT(1),
     EXIST_SW (*)                    CONTROLLED BIT(1),

     1 JLIST (*)                     CONTROLLED,
       3 BLK                         BIT(1),
       3 SIZE                        FIXED BIN(15),
       3 SUBLK_PTR                   POINTER,
     1 ITEM                          BASED (P),
       3 STATE                       FIXED BIN(15),
       3 NEXT                        POINTER;

   CALL COMPUTE_FINAL;

   CALL PREVIOUS_TABLE;

   CALL PARTITION;

   CALL CONST_MINIMAL;

   CALL PRINT_MINIMAL;

   RETURN;
```

```
/*************************************************************/
/*                                                           */
/*              COMPUTE FINAL STATES                         */
/*                                                           */
/*************************************************************/
COMPUTE_FINAL: PROC;

 DCL
     OUT_STRING                          CHAR(250) VARYING,
     I_STRING                            PIC '99',
     (I, J, K, L, #)                     FIXED BIN(15),
     FIND                                BIT(1),
     MUL_FINAL (*,*)                     CTL FIXED BIN(15);

  ALLOCATE BLOCKVEC (0:STATE_NOW) INIT(0);
  ALLOCATE FINAL_NFAPTR(0:STATE_NOW); FINAL_NFAPTR = NULL;

  # = 0;
  DO I = 0 TO STATE_NOW;
     IF FINAL_SW (I) THEN DO;
                          # = # + 1;
                          BLOCKVEC (I) = #;
                          CALL INSERT_LIST (I,I);
                          END;
  END;
  #_OF_BLOCKS = #;

  ALLOCATE MUL_FINAL (STATE_OLD+1:STATE_NOW,0:#_OF_BLOCKS);
           MUL_FINAL = 0;
  DO I = STATE_OLD+1 TO STATE_NOW; /* COL 0 : # OF FINAL */
     DO J = 1 TO STATE_TAB(I,0);
        # = BLOCKVEC (STATE_TAB(I,J));
        IF # ¬= 0 THEN DO;
                       MUL_FINAL(I,0) = MUL_FINAL(I,0) + 1;
                       MUL_FINAL(I,MUL_FINAL(I,0)) = #;
                       CALL INSERT_LIST (I,STATE_TAB(I,J));
                       END;
     END;
  END;

  DO I = STATE_OLD+1 TO STATE_NOW;
     # = MUL_FINAL(I,0);
     IF # = 0 THEN ;                  /* NO FINAL STATE     */
     ELSE IF # = 1 THEN               /* ONE FINAL STATE    */
             BLOCKVEC (I) = MUL_FINAL (I,#);
     ELSE DO;
          FIND = '0'B;
          DO K = STATE_OLD+1 TO I-1 WHILE(¬FIND);
             IF # = MUL_FINAL (K,0) THEN/* COMPARE ROW I */
                DO;                  /* TO ROW K IN MUL_FINAL*/
```

```
                FIND = '1'B;
                DO L = 1 TO # WHILE (FIND);
                   IF MUL_FINAL(I,L) ¬= MUL_FINAL(K,L) THEN
                      FIND = '0'B;
                  END;
                IF FIND THEN
                   BLOCKVEC (I) = BLOCKVEC (K);
                END;
             END;  /* OF K LOOP */
          IF ¬FIND THEN DO;
                        #_OF_BLOCKS = #_OF_BLOCKS + 1;
                        BLOCKVEC (I) = #_OF_BLOCKS;
                        END;
          END;
    END;
 DO I = 0 TO STATE_NOW;
    IF BLOCKVEC(I) > 0 THEN FINAL_SW(I) = '1'B;
    ELSE FINAL_SW(I) = '0'B;
 END;

 PUT SKIP (3) EDIT ('  THE INITIAL PARTITIONS ARE') (A);
 PUT SKIP (2) EDIT ('STATES ---') (A);  OUT_STRING = ' ';
 DO I = 0 TO #_OF_BLOCKS;
    PUT EDIT (' (') (A);        I_STRING = I;
    OUT_STRING = OUT_STRING || I_STRING || ' ';
    DO J = 0 TO STATE_NOW;
       IF BLOCKVEC(J) = I THEN DO;
                               PUT EDIT (J) (F(3));
                               OUT_STRING=OUT_STRING||'   ';
                               END;
    END;
    PUT EDIT (')') (A);
 END;
 PUT SKIP EDIT('BLOCK # --'||OUT_STRING) (A);

 RETURN;

 INSERT_LIST: PROC (I,#);
 DCL  (I, #)                    FIXED BIN(15),
      P                         POINTER;
 ALLOCATE NODE SET (P);
 P->LOC = #;   P->NEXT = FINAL_NFAPTR(I);
 FINAL_NFAPTR (I) = P;
 RETURN;
END INSERT_LIST;
END COMPUTE_FINAL;
```

```
/*************************************************************/
/*                                                           */
/*  INITIALIZATION AND CONSTRUCT PREVIOUS STATE TABLE        */
/*                                                           */
/*************************************************************/
PREVIOUS_TABLE: PROC;

 DCL
   FIND                                    BIT(1),
   P                                       POINTER,
   (I, J, K, A, #, MAX)                    FIXED BIN(15);

   ALLOCATE P_TAB (0:STATE_NOW,#_OF_ALPHA);
            P_TAB = NULL;

   DO I = #_OF_ALPHA TO 1 BY -1;          /* BACKWARD          */
      DO J = STATE_NOW TO 0 BY -1;
         DO K = STATE_NOW TO 0 BY -1;
            ALLOCATE PAIR SET (P);
            P->ROW# = J;
            P->COL# = K;
            # = DFA(I,J,K);
            P->PTR = P_TAB(#,I);         /* INSERT AT FRONT*/
            P_TAB(#,I) = P;
         END;
      END;
   END;

   DO I = 0 TO STATE_NOW;                 /* DETECT UNREACH.*/
      FIND = '0'B;                        /* STATES         */
      DO J = 1 TO #_OF_ALPHA WHILE (¬FIND);
         IF P_TAB(I,J) ¬= NULL THEN FIND = '1'B;
       END;
      IF ¬FIND THEN
         BLOCKVEC (I) = DEAD_STATE;
   END;

   ALLOCATE BLOCKSIZE (0:STATE_NOW) INIT (0);
   DO I = 0 TO STATE_NOW;
      # = BLOCKVEC (I);
      IF # ¬= DEAD_STATE THEN
         BLOCKSIZE (#) = BLOCKSIZE (#) + 1;
   END;

   MAX = 0;                               /*GET MAX BLK SIZE*/
   DO I = 0 TO #_OF_BLOCKS;
      IF MAX < BLOCKSIZE(I) THEN MAX = BLOCKSIZE(I);
   END;

   ALLOCATE L (#_OF_ALPHA,0:STATE_NOW) INIT('0'B);
```

```
   DO I = 0 TO #_OF_BLOCKS;                /* INITIAL L(A)    */
      IF BLOCKSIZE(I) ¬= MAX THEN
         DO J = 1 TO #_OF_ALPHA;
            L (J,I) = '1'B;
         END;
   END;

   ALLOCATE BLK_LIST (0:STATE_NOW,0:STATE_NOW) INIT('0'B);
   ALLOCATE BLK_HDR (2,0:STATE_NOW)     /* 1: ROW, 2: COL */
            INIT('0'B);
   ALLOCATE EXIST_SW (0:STATE_NOW) INIT('0'B);
   ALLOCATE JLIST (0:STATE_NOW);
            JLIST.BLK = '0'B;
            JLIST.SIZE = 0;
            JLIST.SUBLK_PTR = NULL;

   RETURN;
END PREVIOUS_TABLE;
```

```
/*************************************************************/
/*                                                           */
/*     PARTITION UNTIL L CONTAINS NO '1'B                    */
/*                                                           */
/*************************************************************/
PARTITION: PROC;

 DCL
   CONTINUE                          BIT(1),
   (I, A)                            FIXED BIN(15);

  CONTINUE = '1'B;
  DO WHILE (CONTINUE);
     DO I = 0 TO #_OF_BLOCKS;
        DO A = 1 TO #_OF_ALPHA;
           IF L(A,I) THEN
              DO;
              CALL CONST_BLKLIST (A,I);    /* FOR BLOCK I */
              CALL SPLIT (ROW);
              CALL SPLIT (COL);
              L(A,I) = '0'B;
              END;
        END;
     END;

     CONTINUE = '0'B;
     DO I = 0 TO #_OF_BLOCKS;
        DO A = 1 TO #_OF_ALPHA;
           IF L(A,I) THEN CONTINUE = '1'B;
        END;
     END;
  END;

  RETURN;
```

```
/*****************************************************************/
/*                                                               */
/*       CONSTRUCT BLK_LIST FOR BLOCK BLK#                       */
/*                                                               */
/*****************************************************************/
CONST_BLKLIST: PROC (A,BLK#);

 DCL
   (A, BLK#, I, R, C)                    FIXED BIN(15),
   P                                     POINTER;

  BLK_LIST = '0'B;   BLK_HDR = '0'B;

  DO I = 0 TO STATE_NOW;
     IF BLOCKVEC(I) = BLK# THEN
        DO;
        P = P_TAB(I,A);
        DO WHILE (P ¬= NULL);
           R = P->ROW#;
           C = P->COL#;
           BLK_LIST(R,C) = '1'B;
           BLK_HDR(ROW,R) = '1'B;
           BLK_HDR(COL,C) = '1'B;
           P = P->PTR;
         END;
        END;
  END;

  RETURN;
 END CONST_BLKLIST;
```

```
/*************************************************************/
/*                                                           */
/*       SPLIT EACH BLOCK IN JLIST                           */
/*                                                           */
/*************************************************************/
SPLIT: PROC (RC);

 DCL
   (I, J, K, RC, INX)                    FIXED BIN(15),
   T                                     BIT(1),
   P                                     POINTER;

  EXIST_SW = '0'B;
  JLIST.BLK = '0'B; JLIST.SIZE = 0; JLIST.SUBLK_PTR = NULL;
  DO I = 0 TO STATE_NOW;
     IF BLK_HDR(RC,I) THEN
        DO J = 0 TO STATE_NOW;
           IF RC = ROW THEN T = BLK_LIST(I,J);
           ELSE T = BLK_LIST(J,I);
           IF T & ¬EXIST_SW(J) THEN
              CALL ADDJLIST(J);
        END;
  END;

  DO J = 0 TO #_OF_BLOCKS;
     IF JLIST(J).BLK THEN
        IF JLIST(J).SIZE < BLOCKSIZE(J) THEN
           DO;                                /* SPLIT B(J)    */
           BLOCKSIZE(J) = BLOCKSIZE(J) - JLIST(J).SIZE;
           #_OF_BLOCKS = #_OF_BLOCKS + 1;/*NEW SUBLK B(K)*/
           BLOCKSIZE(#_OF_BLOCKS) = JLIST(J).SIZE;
           P = JLIST(J).SUBLK_PTR;
           DO WHILE (P ¬= NULL);
              BLOCKVEC(P->STATE) = #_OF_BLOCKS;
              P = P->NEXT;
             END;
           IF L(A,J) THEN INX = #_OF_BLOCKS;
           ELSE IF BLOCKSIZE(J)<BLOCKSIZE(#_OF_BLOCKS) THEN
                   INX = J;
           ELSE INX = #_OF_BLOCKS;
           DO K = 1 TO #_OF_ALPHA;
              L(K,INX) = '1'B;
             END;
           END;
        ELSE;                      /* |B(K)|=|B(J)|, NO SPLIT*/
     ELSE;                         /* NOT IN JLIST           */
  END;
  RETURN;
END SPLIT;
```

```
/*************************************************************/
/*                                                           */
/*          ADD STATE J TO JLIST                             */
/*                                                           */
/*************************************************************/
ADDJLIST: PROC (J);

 DCL (J, #)                                FIXED BIN(15),
      P                                    POINTER;

   EXIST_SW(J) = '1'B;        /* ADD STATE J TO SUBLK      */
   # = BLOCKVEC(J);           /* GET BLOCK #               */

   IF BLOCKSIZE(#) = 1 THEN RETURN; /* A SINGLE STATE BLK*/
   ALLOCATE ITEM SET (P);
   IF JLIST(#).BLK THEN
        DO;                   /* INSERT STATE J AT FRONT OF*/
        JLIST(#).SIZE = JLIST(#).SIZE+1; /*SUBLK LIST FOR*/
        P->STATE = J;         /*              JLIST(#)     */
        P->NEXT = JLIST(#).SUBLK_PTR;
        JLIST(#).SUBLK_PTR = P;
        END;
   ELSE DO;                   /*CREATE A NEW SUBLK LIST FOR*/
        JLIST(#).BLK = '1'B;/* JLIST(#)                    */
        JLIST(#).SIZE = 1;
        JLIST(#).SUBLK_PTR = P;
        P->STATE = J;
        P->NEXT = NULL;
        END;

   RETURN;
END ADDJLIST;

END PARTITION;
```

```
/*************************************************************/
/*                                                           */
/*        CONSTRUCT A MINIMAL DTA                            */
/*                                                           */
/*************************************************************/
CONST_MINIMAL: PROC;

 DCL   (P#, I, J, K)                    FIXED BIN(15);

  P# = BLOCKVEC(0);          /* LET BLK 0 BE THE BEGINNING*/
  IF P# ¬= DEAD_STATE THEN
     DO I = 0 TO STATE_NOW;          /* STATE              */
        IF BLOCKVEC(I) = P# THEN
           BLOCKVEC(I) = 0;
        ELSE IF BLOCKVEC(I) = 0 THEN
                 BLOCKVEC(I) = P#;
     END;

  ALLOCATE STATEVEC (0:#_OF_BLOCKS);/* SAVE CORR. STATE  */
  DO I = 0 TO #_OF_BLOCKS;          /* FOR EACH BLOCK #  */
     DO J = 0 TO STATE_NOW;
        IF BLOCKVEC(J) = I THEN
           STATEVEC(I) = J;
     END;
  END;

  ALLOCATE MINIMAL(#_OF_ALPHA,0:#_OF_BLOCKS,0:#_OF_BLOCKS);
  DO I = 1 TO #_OF_ALPHA;
     DO J = 0 TO #_OF_BLOCKS;
        DO K = 0 TO #_OF_BLOCKS;
           MINIMAL(I,J,K) = BLOCKVEC(
                    DFA(I,STATEVEC(J),STATEVEC(K)));
        END;
     END;
  END;

  RETURN;
END CONST_MINIMAL;
```

```
/*************************************************************/
/*                                                           */
/*         PRINT MINIMAL TREE AUTOMATON                      */
/*                                                           */
/*************************************************************/
PRINT_MINIMAL: PROC;

 DCL (I, J, K)                              FIXED BIN(15),
     I_STRING                               PIC '29',
     OUT_STRING                             CHAR(250) VARYING;

 PUT SKIP (3) EDIT ('  THE FINAL PARTITIONS ARE')(A);
 PUT SKIP(2) EDIT ('DTA STATES ---')(A); OUT_STRING = ' ';
 DO I = 0 TO #_OF_BLOCKS;
    PUT EDIT (' (') (A);      I_STRING = I;
    OUT_STRING = OUT_STRING || I_STRING || ' ';
    DO J = 0 TO STATE_NOW;
     IF BLOCKVEC(J) = I THEN DO;
      PUT EDIT(J)(F(3)); OUT_STRING=OUT_STRING||'    '; END;
    END;
    PUT EDIT (')') (A);
 END;
 PUT SKIP EDIT('NEW STATES ---'||OUT_STRING) (A);

 PUT SKIP (3) EDIT('  THE MINIMAL TREE AUTOMATON IS') (A);
 DO I = 1 TO #_OF_ALPHA;
    PUT SKIP(2) EDIT(ALPHA(I)) (X(2),A);
    PUT SKIP EDIT('      ') (A);
    DO K = 0 TO #_OF_BLOCKS;
       PUT EDIT ('(',K,')') (A,F(2),A);
    END;
    DO J = 0 TO #_OF_BLOCKS;
       PUT SKIP EDIT('(',J,')') (A,F(2),A);
       DO K = 0 TO #_OF_BLOCKS;
          PUT EDIT (MINIMAL(I,J,K)) (F(4));
       END;
    END;
 END;

 PUT SKIP (3) EDIT ('  THE BEGINNING STATE IS  0') (A);
 PUT SKIP (3) EDIT ('  THE FINAL STATES ARE') (A);
 DO I = 0 TO #_OF_BLOCKS;
    IF FINAL_SW(STATEVEC(I)) THEN
       PUT EDIT (I) (X(1),F(4));
 END;
 RETURN;
 END PRINT_MINIMAL;

 END MINIMIZATION;
```

```
/*************************************************************/
/*************************************************************/
/*                                                           */
/*              DRIVER ROUTINE                               */
/*                                                           */
/*************************************************************/
/*************************************************************/
IDIOM_MATCHING: PROC;

 DCL   X                                    CHAR(LEN_OF_SYMBOL),
       REDUCE_STR                           CHAR(160) VARYING,
       (P_STR, B_STR)                       CHAR(160) VARYING,
       (P, PTR, Q)                          POINTER,
       1 LIST_HEAD                          BASED (Q),
         5 LIST_ID                          FIXED BIN(15),
         5 LIST_PATH_PTR                    POINTER,
         5 LIST_NEXT                        POINTER,
       NONFINAL_TREE (*)                    BIT(1) CONTROLLED,
       (ALPHA_RANGE, K, M, ID#)             FIXED BIN(15),
       (L, R, I, S, ROOT, J, S_BENEFIT)     FIXED BIN(15);

MATCH_PHASE = '1'B;       ALPHA_RANGE = #_OF_ALPHA;
ALLOCATE NONFINAL_TREE (1:#_OF_IDIOMS);

GET SKIP EDIT (X) (A(1));
DO WHILE (X¬=END_OF_IDIOMS);
  B_NODE.MARK,B_NODE.RLINK, B_NODE.STATE, B_NODE.ALPHA_INX,
  B_NODE.LLINK, B_NODE.BENEFIT = 0;     B_NODE.INFO = 0;
  B_NODE.NON_MARK = '0'B; B_NODE.PATH = '';
  B_NODE.PATHLIST_PTR = NULL;

  PUT SKIP (3) EDIT ('  THE INPUT EXPRESSION TREE IS') (A);
  PUT SKIP EDIT (X) (A);
  I_TREE_INX = 0;
  ROOT = NARY_TO_BIN ('',X);
  DO I = ROOT TO I_TREE_INX;
     B_NODE(I).ALPHA_INX = I_TREE(I).ALPHA_INX;
     B_NODE(I).LLINK = I_TREE(I).LLINK;
     B_NODE(I).RLINK = I_TREE(I).RLINK;
     B_NODE(I).PATH = I_TREE(I).PATH;
  END;

  B_NODE_INX = I_TREE_INX;
  B_NODE(0).STATE = 0;              /* LET 0 BE THE BEG STATE*/

  DO I = B_NODE_INX TO ROOT BY -1;   /* CHECK ERROR STATE*/
     IF B_NODE(I).ALPHA_INX > ALPHA_RANGE THEN S = 0;
     ELSE DO;
          L = B_NODE(B_NODE(I).LLINK).STATE;
          R = B_NODE(B_NODE(I).RLINK).STATE;
```

```
            S = MINIMAL(B_NODE(I).ALPHA_INX,L,R);
            END;
         B_NODE(I).STATE = S;

         J = STATEVEC(S);   NONFINAL_TREE = '0'B;
         IF FINAL_SW(J) THEN
            CALL MARK_IDIOM (I,J);
         DO K = 1 TO STATE_TAB(J,0);
            M = STATE_TAB(J,K);
            IF FINAL_SW (M) THEN ;
            ELSE IF NONFINAL_CLOSURE_SW (M) &
                    ¬ NONFINAL_TREE(FINAL_ID#(M)) THEN DO;
                    PTR =NULL;        ID# = FINAL_ID# (M);
                    P = FINAL_PATH_INFO(M).PRODUCT_PTR;
                    CALL SUM_BENEFIT (I,P,1,ID#);
                    P = FINAL_PATH_INFO(M).CLOSURE_PTR;
                    CALL SUM_BENEFIT (I,P,2,ID#);
                    P = FINAL_PATH_INFO(M).VAR_PTR;
                    CALL SUM_BENEFIT (I,P,3,ID#);
                    IF PTR ¬= NULL THEN DO;
                       ALLOCATE LIST_HEAD SET (Q);
                       Q->LIST_PATH_PTR = PTR;
                       Q->LIST_ID = ID#;
                       Q->LIST_NEXT = B_NODE(I).PATHLIST_PTR;
                       B_NODE(I).PATHLIST_PTR = Q;
                       NONFINAL_TREE (ID#) = '1'B;
                       END;
                    END;
            END;
      END;

      PUT SKIP (3) EDIT ('  THE BEST MATCHES ARE') (A);
      PUT SKIP EDIT ('INPUT ---') (A);
      REDUCE_STR = 'OUTPUT --';      /* OUTPUT FOR REDUCED EXP*/

      B_NODE(ROOT).NON_MARK = '1'B; /* MARK EACH NODE WHICH  */
      CALL SELECT_IDIOM (ROOT);     /* IS FRON OF IDIOMS     */
      P_STR = L_PARNTHE || R_PARNTHE;   B_STR = BLANK || BLANK;
      DO WHILE (LENGTH(P_STR)<=LENGTH(REDUCE_STR));
         I = INDEX (REDUCE_STR,P_STR);
         IF I>0 THEN SUBSTR(REDUCE_STR,I,LENGTH(P_STR))=B_STR;
         P_STR = SUBSTR(P_STR,1,1) || BLANK || SUBSTR(P_STR,2);
         B_STR = B_STR || BLANK;
      END;
      PUT SKIP EDIT (REDUCE_STR) (A);

      GET SKIP EDIT(X) (A(1));
   END;
   RETURN;
```

```
/*************************************************************/
/*                                                           */
/*      EACH NODE HAS ONLY ONE MARKED IDIOM                  */
/*                                                           */
/*************************************************************/
  MARK_IDIOM: PROC (I,J);

 DCL   (I, J, S, MAX, ID#)                FIXED BIN(15),
       (P, Q, R)                          POINTER;

  MAX = 0;
  P = FINAL_NFAPTR (J);
  DO WHILE (P¬= NULL);
     S = P->LOC;  S_BENEFIT = FINAL_BENEFIT(S);
     PTR = NULL;  ID# = FINAL_ID# (S);
     Q = FINAL_PATH_INFO(S).PRODUCT_PTR;
     CALL SUM_BENEFIT(I,Q,1,ID#);
     Q = FINAL_PATH_INFO(S).CLOSURE_PTR;
     CALL SUM_BENEFIT(I,Q,2,ID#);
     Q = FINAL_PATH_INFO(S).VAR_PTR;
     CALL SUM_BENEFIT(I,Q,3,ID#);
     IF S_BENEFIT > MAX THEN
        DO;
        MAX = S_BENEFIT;
        B_NODE(I).MARK = ID#;
        ALLOCATE LIST_HEAD SET (R);
        R->LIST_PATH_PTR = PTR;
        R->LIST_ID = ID#;
        B_NODE(I).BENEFIT = MAX;
        END;
     P = P->NEXT;
  END;

  R->LIST_NEXT = B_NODE(I).PATHLIST_PTR;
  B_NODE(I).PATHLIST_PTR = R;
  RETURN;

END MARK_IDIOM;
```

```
/*************************************************************/
/*     ACCUMULATE BENEFIT FUNCTION VALUES AND PATH LIST      */
/*************************************************************/
SUM_BENEFIT: PROC (I,P,TYPE,ID#);

DCL   (I, J, K, ID#)          FIXED BIN(15),
      (P, Q, R, QQ)           POINTER,
       FIND                   BIT(1),
       TYPE                   FIXED BIN(15);

 DO WHILE (P¬=NULL);
    K = I;  DO J = 1 TO P->P_PATH_LEN;
               IF SUBSTR(P->P_PATH,J,1) THEN
                     K = B_NODE(K).RLINK;
               ELSE K = B_NODE(K).LLINK;
               IF K = 0 THEN RETURN;
            END;
    IF TYPE = 1 | TYPE = 3 |
       P->P_INX = B_NODE(K).ALPHA_INX THEN
       DO;
       QQ = B_NODE(K).PATHLIST_PTR;
       FIND = '0'B;
       IF QQ¬=NULL & TYPE ¬= 3 THEN
          DO WHILE (¬FIND & QQ¬=NULL);
             IF QQ->LIST_ID ¬= ID# THEN
                QQ = QQ->LIST_NEXT;
             ELSE DO;
                  FIND = '1'B;
                  Q = QQ->LIST_PATH_PTR;
                  DO WHILE (Q¬=NULL);
                     ALLOCATE PATH_NODE1 SET (R);
                     R->P_PATH1 = SUBSTR(P->P_PATH,1,
                              P->P_PATH_LEN)||SUBSTR(
                              Q->P_PATH1,1,Q->P_PATH_LEN1);
                     R->P_PATH_LEN1 = P->P_PATH_LEN +
                            Q->P_PATH_LEN1;
                     R->P_NEXT1 = PTR;
                     PTR = R;
                     Q = Q->P_NEXT1;
                   END;
                  S_BENEFIT = S_BENEFIT+B_NODE(K).BENEFIT;
                  END;
          END;
       IF ¬ FIND THEN    DO;
                         ALLOCATE PATH_NODE1 SET (R);
                         R->P_PATH1 = P->P_PATH;
                         R->P_PATH_LEN1 = P->P_PATH_LEN;
                         R->P_NEXT1 = PTR;
                         PTR = R;
                         END;
```

```
      END;
   P = P->P_NEXT;
END;
RETURN;
END SUM_BENEFIT;

/*************************************************************/
/*                                                           */
/*     SELECT THE BEST NON-OVERLAPPED MATCHES                */
/*                                                           */
/*************************************************************/
 SELECT_IDIOM: PROC (I) RECURSIVE;

DCL  (I, J, K, L, R)                FIXED BIN(15),
      SYMBOL                        CHAR(LEN_OF_SYMBOL),
      PARNTHE_SW                    BIT(1),
      #                             PIC '9V',
      (P, Q)                        POINTER;

 L = B_NODE(I).LLINK;     R = B_NODE(I).RLINK;
 PARNTHE_SW = '1'B;

 SYMBOL = ALPHA(B_NODE(I).ALPHA_INX);
 PUT EDIT (SYMBOL||BLANK) (A);
 IF ¬B_NODE(I).NON_MARK  THEN DO;
      REDUCE_STR = REDUCE_STR || BLANK || BLANK;
      PARNTHE_SW = '0'B;
      END;
 ELSE IF B_NODE(I).MARK = 0  THEN DO;
      REDUCE_STR = REDUCE_STR || SYMBOL || BLANK;
      B_NODE(L).NON_MARK = '1'B;
      IF L ¬= 0 THEN DO;
         K = B_NODE(L).RLINK;
         DO WHILE (K¬=0);
            B_NODE(K).NON_MARK = '1'B;
            K = B_NODE(K).RLINK;
            END;
         END;
      END;
 ELSE DO;
      # = B_NODE(I).MARK;
      REDUCE_STR = REDUCE_STR || # || BLANK;
      Q = B_NODE(I).PATHLIST_PTR;
      DO WHILE (Q->LIST_ID ¬= #);
         Q = Q->LIST_NEXT;
      END;
      P = Q->LIST_PATH_PTR;
      DO WHILE (P¬=NULL);
         K = I;  DO J = 1 TO P->P_PATH_LEN1;
                    IF SUBSTR(P->P_PATH1,J,1) THEN
```

```
                        K = B_NODE(K).RLINK;
                      ELSE K = B_NODE(K).LLINK;
                    END;
            B_NODE(K).NON_MARK = '1'B;
            P = P->P_NEXT1;
          END;
        END;

    IF L ¬= 0 THEN DO;
        PUT EDIT (L_PARNTHE) (A);
        IF PARNTHE_SW THEN REDUCE_STR = REDUCE_STR ||
                                        L_PARNTHE;
        ELSE REDUCE_STR = REDUCE_STR || BLANK;
        CALL SELECT_IDIOM (L);
        PUT EDIT (R_PARNTHE||BLANK) (A);
        IF PARNTHE_SW THEN
           REDUCE_STR = REDUCE_STR||R_PARNTHE||BLANK;
        ELSE REDUCE_STR = REDUCE_STR||BLANK||BLANK;
        END;

    IF R ¬= 0 THEN CALL SELECT_IDIOM (R);

    RETURN;
   END SELECT_IDIOM;

   END IDIOM_MATCHING;
  END IDIOM_BIN;
//LKED.SYSLMOD DD
DSN=F.I4983.IDIOM.LIB(BINEW),DISP=(OLD,KEEP),
//   SPACE=(TRK,(24,2,7)),VOL=SER=LIB002,UNIT=DISK
//GO.SYSIN  DD  *
G(/(=(VV),))
,(V,)
,(VV)
$
1#,(  2*,#, 3) ;
$
G(/(=(//),))
G(/(=(//)=))
G(/(=(//),(==)))
G(/(=(,,),(,,)))
G(/(=(,,),(,,(,,))))
G(/(=(,,),(,,(,,(/=)))))
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
/*
//
```

```
/*****************************************************/
/*     Output from the program                       */
/*****************************************************/
```

INPUT INDIVIDUAL TREES ARE

1. G(/(=(VV),))
2. ,(V,)
3. ,(VV)

INPUT IDIOMS ARE

1. 1 #, (2 *, #, 3) ;

THE DETERMINISTIC TREE AUTOMATON IS

G

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(0)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(1)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(2)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(3)	4	4	4	4	4	4	4	13	13	13	13	13	4	4
(4)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(5)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(6)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(7)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(8)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(9)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(10)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(11)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(12)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(13)	0	0	0	0	0	0	0	6	6	6	6	6	0	0

/

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(0)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(1)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(2)	3	0	0	0	0	0	0	6	6	6	6	6	0	0
(3)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(4)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(5)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(6)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(7)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(8)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(9)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(10)	0	0	0	0	0	0	0	6	6	6	6	6	0	0

(11)	0	0	0	0	0	0	0	6	6	6	6	6	0	0
(12)	3	0	0	0	0	0	0	6	6	6	6	6	0	0
(13)	0	0	0	0	0	0	0	6	6	6	6	6	0	0

=

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(0)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(1)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(2)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(3)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(4)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(5)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(6)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(7)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(8)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(9)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(10)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(11)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(12)	0	0	0	0	0	0	0	12	12	12	12	12	0	0
(13)	0	0	0	0	0	0	0	12	12	12	12	12	0	0

,

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(0)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(1)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(2)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(3)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(4)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(5)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(6)	10	10	10	10	10	10	10	11	11	11	11	11	10	10
(7)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(8)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(9)	10	10	10	10	10	10	10	11	11	11	11	11	10	10
(10)	8	8	8	8	8	8	8	9	9	9	9	9	8	8
(11)	10	10	10	10	10	10	10	11	11	11	11	11	10	10
(12)	10	10	10	10	10	10	10	11	11	11	11	11	10	10
(13)	10	10	10	10	10	10	10	11	11	11	11	11	10	10

THE STATES IN DFA ARE

< 0>---{ (0) }
< 1>---{ (1) }
< 2>---{ (2) }
< 3>---{ (3) }
< 4>---{ (4) }
< 5>---{ (5) }
< 6>---{ (6) }

```
<   7>---{ ( 7) }
<   8>---{ ( 8) }
<   9>---{ ( 6),( 8) }
<  10>---{ ( 7),( 8) }
<  11>---{ ( 6),( 7),( 8) }
<  12>---{ ( 2),( 6) }
<  13>---{ ( 4),( 6) }

  THE FINAL STATES ARE ( 4)

  THE INITIAL PARTITIONS ARE

STATES --- (  0  1  2  3  5  6  7  8  9 10 11 12) (  4 13)
BLOCK # --  0                                        1

  THE FINAL PARTITIONS ARE

DTA STATES --- (  0) (  4 13) (  3) (  8  9 10 11) ( 12) (  6)
NEW STATES ---  0      1        2      3             4      5

  THE MINIMAL TREE AUTOMATON IS

  G
      ( 0) ( 1) ( 2) ( 3) ( 4) ( 5)
( 0)   0    0    0    5    0    0
( 1)   0    0    0    5    0    0
( 2)   1    1    1    1    1    1
( 3)   0    0    0    5    0    0
( 4)   0    0    0    5    0    0
( 5)   0    0    0    5    0    0

  /
      ( 0) ( 1) ( 2) ( 3) ( 4) ( 5)
( 0)   0    0    0    5    0    0
( 1)   0    0    0    5    0    0
( 2)   0    0    0    5    0    0
( 3)   0    0    0    5    0    0
( 4)   2    0    0    5    0    0
( 5)   0    0    0    5    0    0

  =
      ( 0) ( 1) ( 2) ( 3) ( 4) ( 5)
( 0)   0    0    0    4    0    0
( 1)   0    0    0    4    0    0
( 2)   0    0    0    4    0    0
( 3)   0    0    0    4    0    0
```

```
( 4)    0    0    0    4    0    0
( 5)    0    0    0    4    0    0

  ,
     ( 0) ( 1) ( 2) ( 3) ( 4) ( 5)
( 0)    3    3    3    3    3    3
( 1)    3    3    3    3    3    3
( 2)    3    3    3    3    3    3
( 3)    3    3    3    3    3    3
( 4)    3    3    3    3    3    3
( 5)    3    3    3    3    3    3

  THE BEGINNING STATE IS   0

  THE FINAL STATES ARE     1


  THE INPUT EXPRESSION TREE IS
G(/(=(//),))

  THE BEST MATCHES ARE
INPUT ---G (/ (= (/ / ) , ) )
OUTPUT --1 (      / /   ,   )


  THE INPUT EXPRESSION TREE IS
G(/(=(//)=))

  THE BEST MATCHES ARE
INPUT ---G (/ (= (/ / ) = ) )
OUTPUT --G (/ (= (/ / ) = ) )


  THE INPUT EXPRESSION TREE IS
G(/(=(//),(==)))

  THE BEST MATCHES ARE
INPUT ---G (/ (= (/ / ) , (= = ) ) )
OUTPUT --1 (      / /       = =     )


  THE INPUT EXPRESSION TREE IS
G(/(=(,,),(,,)))
```

```
  THE BEST MATCHES ARE
INPUT ---G (/ (= (, , ) , (, , ) ) )
OUTPUT --1 (      , ,      , ,     )


  THE INPUT EXPRESSION TREE IS
G(/(=(,,),(,,(,,))))

  THE BEST MATCHES ARE
INPUT ---G (/ (= (, , ) , (, , (, , ) ) ) )
OUTPUT --1 (      , ,      ,    , ,       )


  THE INPUT EXPRESSION TREE IS
G(/(=(,,),(,,(,,(/=)))))

  THE BEST MATCHES ARE
INPUT ---G (/ (= (, , ) , (, , (, , (/ = ) ) ) ) )
OUTPUT --1 (      , ,      ,    ,    / =         )
```